# The Boost C++ Libraries

Introduces 72 libraries with more than 430 examples

*My go-to book for introducing the Boost Libraries*
— Jon Kalb, Boost Conference Chair

Second Edition



Boris Schäling

# 目錄

# Boost C++ 库

来源：Boost C++ 库

版本：1.0 / 19 八月 2010

# Boost C++ 库

## 目录







本书的中文版由 Albert Lai, Jerry Guo, Kun Zeng, Liangfu Chen (主页), Cui Wei 和 Rui Shi 翻译。

# 内容

## 你将学到些什么

本书是对 Boost C++ 库的介绍，Boost 库通过加入一些在实践中非常有用的函数对 C++ 标准进行了补充。 由于 Boost C++ 库是基于 C++ 标准的，所以它们是使用最先进的 C++ 来实现的。 它们是平台独立的，并由于有一个大型的开发人员社区，它可以被包括 Windows 和 Linux 在内的许多操作系统所支持。

Boost C++ 库可以提升你作为一个 C++ 开发人员的生产力。 例如，你可以从智能指针中受益，帮助你写出更可靠的代码，或者使用某个库来开发平台独立的网络应用。 因为多数 Boost C++ 库正被收录进下一个版本的 C++ 标准，所以你可以从今天就开始作好准备。

# 要求

## 你应该懂得些什么

因为 Boost C++ 库是基于且扩展了 C++ 标准，所以你应该懂得 C++ 标准。 你应该了解且能够使用容器、迭代器及算法，最好有听说过以下概念：RAII，函数对象，或是谓词。 你越是了解 C++ 标准，就越能从 Boost C++ 库中受益。

# 第 1 章 简介

## 目录






## 1.1. C++ 与 Boost

Boost C++ 库 是一组基于C++标准的现代库。 其源码按 Boost Software License 来发布，允许任何人自由地使用、修改和分发。 这些库是平台独立的，且支持大多数知名和不那么知名的编译器。

Boost 社区负责开发和发布 Boost C++ 库。 社区由一个很大的C++开发人员群组组成，这些开发人员来自于全球，他们通过网站 www.boost.org 以及几个邮件列表相互协调。 社区的使命是开发和收集高质量的库，作为C++标准的补充。 那些被证实有价值且对于C++应用开发非常重要的库，将会有很大机会在某天被纳入C++标准中。

Boost 社区在1998年左右出现，当时刚刚发布了C++标准的第一个版本。 从那时起，社区就不断地扩大，现在已成为C++标准化工作中的一个重要角色。 虽然 Boost 社区与标准化委员会之间没有直接的关系，但有部分开发者同时活跃于两方。 下一个版本的C++标准很大可能在2011年通过，其中将扩展一批库，这些库均起源于 Boost 社区。

要增强C++项目的生产力，除了C++标准以外，Boost C++ 库是一个不错的选择。 由于当前版本的C++标准在2003年修订之后，C++又有了新的发展，所以 Boost C++ 库提供了许多新的特性。 由于有了 Boost C++ 库，我们无需等待下一个版本的C++标准，就可以立即享用C++演化中取得的最新进展。

Boost C++ 库具有良好的声誉，这基于它们的使用已被证实是非常有价值的。 在面试中询问关于 Boost C++ 库的知识是不常见的，因为知道这些库的开发人员通常也清楚C++的最新创新，并且能够编写和理解现代的C++代码。

## 1.2. 开发过程

正是因为大量的独立开发者和组织的支持和参与，才使用 Boost C++ 库的开发成为可能。 由于 Boost 只接受满足以下条件的库：解决了真实存在的问题、表现出令人信服的设计、使用现代C++来开发且以可理解的方式提供文档，所以每一个 Boost C++ 库的背后都有大量的工作。

C++ 开发者都可以加入到 Boost 社区中，并提出自己的新库。 但是，要将一个想法变成一个 Boost C++ 库，需要投入大量的时间和努力。 其中最重要的是在 Boost 邮件列表中与其他开发者及潜在用户讨论需求和可能的解决方案。

除了这些好象不知从何处冒出来的新库以外，也可以提名一些已有的 C++ 库进入 Boost。 不过，由于对这些库的要求是与专门为 Boost 开发的库一样的，所以可能需要对它们进行大量的修改。

一个库是否被接纳入 Boost，取决于评审过程的结果。 库的开发者可以申请评审，这通常需要10天的时间。 在这段时间内，其他开发者被邀请对这个库进行评分。 基于正面和负面评价的数量，评审经理将决定该库是否被接纳进入 Boost。 由于有些开发者是在评审阶段才首次公开库的代码，所以在评审期间被要求对库进行修改并不罕见。

如果一个库是因为技术原因被拒绝，那么它还有可能在修改之后对更新后的版本申请新的评审。 但是，如果一个库是因为不能解决实际问题或未能提供令人信服的解决方案而被拒绝，那么再一次评审也很可能会被拒绝。

因为可能随时接纳新的库，所以 Boost C++ 库会每三个月发布一次新版本。本书所涉及的库均基于2010年2月发布的 1.42.0 版本。

请注意，另外还有一些库已被接纳，但尚未成为 Boost C++ 库发布版的一部分。在被包含进发布版之前，它们必须手工安装。

## 1.3. 安装

Boost C++ 库均带有源代码。其中大多数库只包含头文件，可以直接使用，但也有一些库需要编译。 为了尽可能容易安装，可以使用 Boost Jam 进行自动安装。 无需逐个库进行检查和编译，Boost Jam 自动安装整个库集。 它支持许多操作系统和编译器，并且知道如何基于适当的配置文件来编译单个库。

为了在 Boost Jam 的帮助下自动安装，要使用一个名为 **bjam** 的应用程序，它也带有源代码。 对于某些操作系统，包括 Windows 和 Linux，也有预编译好的 **bjam** 二进制文件。

为了编译 **bjam** 本身，要执行一个名为 **build** 的简单脚本，它也为不同的操作系统提供了源代码。 对于 Windows，它是批处理文件 `build.bat` 。 对于 Linux，文件名为 `build.sh` 。

如果执行 **build** 时不带任何命令行选项，则该脚本尝试找到一个合适的编译器来生成 **bjam**。 通过使用命令行开关，称为 toolset，可以选择特定的编译器。 对于 Windows，**build** 支持 toolsets `vc7` , `vc8` 和 `vc9` ，可以选择不同版本的 Microsoft C++ 编译器。 要从 Visual Studio 2008 的C++编译器编译 **bjam**，需要指定命令 **build vc9**。对于 Linux，支持 toolsets `gcc` 和 `intel-linux` ，分别选定 GCC 和 Intel 的C++编译器。

应用程序 **bjam** 必须复制到本地的 Boost 目录 - 不论它是编译出来的还是下载的预编译二进制文件。 然后就可以不带任何命令行选项地执行 **bjam**，编译并安装 Boost C++ 库。 由于缺省选项 - 在这种情况下所使用的 - 并不一定是最好的选择，

所以以下列出最重要的几个选项供参考：

- 声明 `stage` 或 `install` 可以指定 Boost C++ 库是安装在一个名为 `stage` 的子目录下，还是在系统范围内安装。 "系统范围"的意义取决于操作系统。 在 Windows 中，目标目录是 `C:\Boost` ；而在 Linux 中则是 `/usr/local` 。 目标目录也可以用 `--prefix` 选项明确指出。
- 如果不带任何命令行选项执行 **bjam**，则它会自己搜索一个合适的C++编译器。 可以通过 `--toolset` 选项来指定一个特定的编译器。 要在 Windows 中指定 Visual Studio 2008 的 Microsoft C++ 编译器，**bjam** 执行时要带上 `--toolset=msvc-9.0` 选项。 要在 Linux 中指定 GCC 编译器，则要给出 `--toolset=gcc` 选项。
- 命令行选项 `--build-type` 决定了创建何种方式的库。 缺省情况下，该选项设为 `minimal` ，即只创建发布版。 对于那些想用 Visual Studio 或 GCC 构建他们项目的调试版的开发者来说，可能是一个问题。 因为这些编译器会自动尝试链接调试版的 Boost C++ 库，这样就会给出一个错误信息。 在这种情况下，应将 `--build-type` 选项设为 `complete` ，以同时生成 Boost C++ 库的调试版和发布版，当然，所需时间也会更长一些。

要用 Visual Studio 2008 的C++编译器同时生成 Boost C++ 库的调试版和发布版，并将它们安装在目录 `D:\Boost` 中，应执行的命令是 **bjam --toolset=msvc-9.0 --build-type=complete --prefix=D:\Boost install**. 要在 Linux 中使用缺省目录创建它们，则要执行的命令是 **bjam --toolset=gcc --build-type=complete install**.

其它多个命令行选项可用于指定如何编译 Boost C++ 库的一些细节设定。 我通常在 Windows 下使用以下命令：**bjam --toolset=msvc-9.0 debug release link=static runtime-link=shared install**. `debug` 和 `release` 使得调试版和发布版均被生成。 `link=static` 则只创建静态库。 `runtime-link=shared` 则是指定 C++ 运行时库是动态链接的，这是在 Visual Studio 2008 中对C++项目的缺省设置。

## 1.4. 概述

Boost C++ 库的 1.42.0 版本包含了超过90个库，本书只详细讨论了以下各库：

表 1.1. 讨论到的库

| Boost C++ 库 | C++ 标准 | 简要说明 |
| --- | --- | --- |
| Boost.Any | | Boost.Any 提供了一个名为 `boost::any` 的数据类型，可以存放任意的类型。 例如，一个类型为 `boost::any` 的变量可以先存放一个 `int` 类型的值，然后替换为一个 `std::string` 类型的字符串。 |
| Boost.Array | TR1 | Boost.Array 可以把 C++ 数组视同 C++ |

| | | |
|---|---|---|
| Boost.Array | TR1 | 标准的容器。 |
| Boost.Asio | TR2 | Boost.Asio 可用于开发异步处理数据的应用，如网络应用。 |
| Boost.Bimap | | Boost.Bimap 提供了一个名为 `boost::bimap` 的类，它类似于 `std::map` . 主要的差别在于 `boost::bimap` 可以同时从键和值进行搜索。 |
| Boost.Bind | TR1 | Boost.Bind 是一种适配器，可以将函数作为模板参数，即使该函数的签名与模板参数不兼容。 |
| Boost.Conversion | | Boost.Conversion 提供了三个转型操作符，分别执行向下转型、交叉转型，以及不同数字类型间的值转换。 |
| Boost.DateTime | | Boost.DateTime 可用于以灵活的格式处理、读入和写出日期及时间值。 |
| Boost.Exception | | Boost.Exception 可以在抛出的异常中加入额外的数据，以便在 `catch` 处理中提供更多的信息。 这有助于更容易地调试，以及对异常情况更好地作出反应。 |
| Boost.Filesystem | TR2 | Boost.Filesystem 提供了一个类来处理路径信息，还包含了几个访问文件和目录的函数。 |
| Boost.Format | | Boost.Format 以一个类型安全且可扩展的 `boost::format` 类替代了 `std::printf()` 函数。 |
| Boost.Function | TR1 | Boost.Function 简化了函数指针的定义。 |
| Boost.Interprocess | | Boost.Interprocess 允许多个应用通过共享内存以快速、高效的方式进行通信。 |
| Boost.Lambda | | Boost.Lambda 可以定义匿名的函数。 代码被内联地声明和执行，避免了单独的函数调用。 |
| Boost.Multiindex | | Boost.Multiindex 定义了一些新的容器，它们可以同时支持多个接口，如 `std::vector` 和 `std::map` 的接口。 |

| | | |
|---|---|---|
| Boost.NumericConversion | | Boost.NumericConversion 提供了一个转型操作符，可以安全地在不同的数字类型间进行值转换，不会生成上溢出或下溢出的条件。 |
| Boost.PointerContainer | | Boost.PointerContainer 提供了专门为动态分配对象进行优化的容器。 |
| Boost.Ref | TR1 | Boost.Ref 的适配器可以将不可复制对象的引用传给需要复制的函数。 |
| Boost.Regex | TR1 | Boost.Regex 提供了通过正则表达式进行文本搜索的函数。 |
| Boost.Serialization | | 通过 Boost.Serialization，对象可以被序列化，如保存在文件中，并在以后重新导入。 |
| Boost.Signals | | Boost.Signal 是一个事件处理的框架，基于所谓的 signal/slot 概念。 函数与信号相关联并在信号被触发时自动被调用。 |
| Boost.SmartPoiners | TR1 | Boost.SmartPoiners 提供了多个智能指针，简化了动态分配对象的管理。 |
| Boost.Spirit | | Boost.Spirit 可以用类似于 EBNF (扩展巴科斯范式)的语法生成词法分析器。 |
| Boost.StringAlgorithms | | Boost.StringAlgorithms 提供了多个独立的函数，以方便处理字符串。 |
| Boost.System | TR2 | Boost.System 提供了一个处理系统相关或应用相关错误代码的框架。 |
| Boost.Thread | C++0x | Boost.Thread 可用于开发多线程应用。 |
| Boost.Tokenizer | | Boost.Tokenizer 可以对一个字符串的各个组件进行迭代。 |
| Boost.Tuple | TR1 | Boost.Tuple 提供了泛化版的 `std::pair` ，可以将任意数量的数据组在一起。 |
| Boost.Unordered | TR1 | Boost.Unordered 扩展了 C++ 标准的容器，增加了 `boost::unordered_set` 和 `boost::unordered_map` . |
| Boost.Variant | | Boost.Variant 可以定义多个数据类型，类似于 `union` , 将多个数据类型组在一起。 Boost.Variant 比 `union` 优胜的地方在于它可以使用类。 |

Technical Report 1 是在2003年发布的，有关 C++0x 标准和 Technical Report 2 的一些细节才能反映当前的状态。由于无论是下一个版本的 C++ 标准，还是 Technical Report 2 都尚未被批准，所以在往后的时间里，它们仍然可能会有改变。

# 第 2 章 智能指针

目录






## 2.1. 概述

1998年修订的第一版C++标准只提供了一种智能指针： `std::auto_ptr` 。 它基本上就像是个普通的指针： 通过地址来访问一个动态分配的对象。 `std::auto_ptr` 之所以被看作是智能指针，是因为它会在析构的时候调用 `delete` 操作符来自动释放所包含的对象。 当然这要求在初始化的时候，传给它一个由 `new` 操作符返回的对象的地址。 既然 `std::auto_ptr` 的析构函数会调用 `delete` 操作符，它所包含的对象的内存会确保释放掉。 这是智能指针的一个优点。

当和异常联系起来时这就更加重要了：没有 `std::auto_ptr` 这样的智能指针，每一个动态分配内存的函数都需要捕捉所有可能的异常，以确保在异常传递给函数的调用者之前将内存释放掉。 Boost C++ 库 Smart Pointers 提供了许多可以用在各种场合的智能指针。

## 2.2. RAII

智能指针的原理基于一个常见的习语叫做 RAII ：资源申请即初始化。 智能指针只是这个习语的其中一例——当然是相当重要的一例。 智能指针确保在任何情况下，动态分配的内存都能得到正确释放，从而将开发人员从这项任务中解放了出来。 这包括程序因为异常而中断，原本用于释放内存的代码被跳过的场景。 用一个动态分配的对象的地址来初始化智能指针，在析构的时候释放内存，就确保了这一点。 因为析构函数总是会被执行的，这样所包含的内存也将总是会被释放。

无论何时，一定得有第二条指令来释放之前另一条指令所分配的资源时，RAII 都是适用的。 许多的 C++ 应用程序都需要动态管理内存，因而智能指针是一种很重要的 RAII 类型。 不过 RAII 本身是适用于许多其它场景的。

```
#include <windows.h>

class windows_handle
{
  public:
    windows_handle(HANDLE h)
      : handle_(h)
    {
    }

    ~windows_handle()
    {
      CloseHandle(handle_);
    }

    HANDLE handle() const
    {
      return handle_;
    }

  private:
    HANDLE handle_;
};

int main()
{
  windows_handle h(OpenProcess(PROCESS_SET_INFORMATION, FALSE, GetC
  SetPriorityClass(h.handle(), HIGH_PRIORITY_CLASS);
}
```

- 下载源代码

上面的例子中定义了一个名为 `windows_handle` 的类，它的析构函数调用了 `CloseHandle()` 函数。 这是一个 Windows API 函数，因而这个程序只能在 Windows 上运行。 在 Windows 上，许多资源在使用之前都要求打开。 这暗示着一旦资源不再使用之后就应该关闭。 `windows_handle` 类的机制能确保这一点。

`windows_handle` 类的实例以一个句柄来初始化。 Windows 使用句柄来唯一的标识资源。 比如说， `OpenProcess()` 函数返回一个 `HANDLE` 类型的句柄，通过该句柄可以访问当前系统中的进程。 在示例代码中，访问的是进程自己——换句话说就是应用程序本身。

我们通过这个返回的句柄提升了进程的优先级，这样它就能从调度器那里获得更多的 CPU 时间。 这里只是用于演示目的，并没什么实际的效应。 重要的一点是：通过 `OpenProcess()` 打开的资源不需要显示的调用 `CloseHandle()` 来关闭。 当

然，应用程序终止时资源也会随之关闭。 然而，在更加复杂的应用程序里， `windows_handle` 类确保当一个资源不再使用时就能正确的关闭。 某个资源一旦离开了它的作用域——上例中 `h` 的作用域在 `main()` 函数的末尾——它的析构函数会被自动的调用，相应的资源也就释放掉了。

## 2.3. 作用域指针

一个作用域指针独占一个动态分配的对象。 对应的类名为 `boost::scoped_ptr` ，它的定义在 `boost/scoped_ptr.hpp` 中。 不像 `std::auto_ptr` ，一个作用域指针不能传递它所包含的对象的所有权到另一个作用域指针。 一旦用一个地址来初始化，这个动态分配的对象将在析构阶段释放。

因为一个作用域指针只是简单保存和独占一个内存地址，所以 `boost::scoped_ptr` 的实现就要比 `std::auto_ptr` 简单。 在不需要所有权传递的时候应该优先使用 `boost::scoped_ptr` 。 在这些情况下，比起 `std::auto_ptr` 它是一个更好的选择，因为可以避免不经意间的所有权传递。

```
#include <boost/scoped_ptr.hpp>

int main()
{
  boost::scoped_ptr<int> i(new int);
  *i = 1;
  *i.get() = 2;
  i.reset(new int);
}
```

- 下载源代码

一经初始化，智能指针 `boost::scoped_ptr` 所包含的对象，可以通过类似于普通指针的接口来访问。 这是因为重载了相关的操作符 `operator*()` ， `operator->()` 和 `operator bool()` 。 此外，还有 `get()` 和 `reset()` 方法。 前者返回所含对象的地址，后者用一个新的对象来重新初始化智能指针。 在这种情况下，新创建的对象赋值之前会先自动释放所包含的对象。

`boost::scoped_ptr` 的析构函数中使用 `delete` 操作符来释放所包含的对象。 这对 `boost::scoped_ptr` 所包含的类型加上了一条重要的限制。 `boost::scoped_ptr` 不能用动态分配的数组来做初始化，因为这需要调用 `delete[]` 来释放。 在这种情况下，可以使用下面将要介绍的 `boost:scoped_array` 类。

## 2.4. 作用域数组

作用域数组的使用方式与作用域指针相似。 关键不同在于，作用域数组的析构函数使用 `delete[]` 操作符来释放所包含的对象。 因为该操作符只能用于数组对象，所以作用域数组必须通过动态分配的数组来初始化。

对应的作用域数组类名为 `boost::scoped_array` ，它的定义在 `boost/scoped_array.hpp` 里。

```
#include <boost/scoped_array.hpp>

int main()
{
  boost::scoped_array<int> i(new int[2]);
  *i.get() = 1;
  i[1] = 2;
  i.reset(new int[3]);
}
```

- 下载源代码

`boost:scoped_array` 类重载了操作符 `operator[]()` 和 `operator bool()` 。 可以通过 `operator[]()` 操作符访问数组中特定的元素，于是 `boost::scoped_array` 类型对象的行为就酷似它所含的数组。

正如 `boost::scoped_ptr` 那样， `boost:scoped_array` 也提供了 `get()` 和 `reset()` 方法，用来返回和重新初始化所含对象的地址。

## 2.5. 共享指针

这是使用率最高的智能指针，但是 C++ 标准的第一版中缺少这种指针。 它已经作为技术报告1（TR 1）的一部分被添加到标准里了。 如果开发环境支持的话，可以使用 `memory` 中定义的 `std::shared_ptr` 。 在 Boost C++ 库里，这个智能指针命名为 `boost::shared_ptr` ，定义在 `boost/shared_ptr.hpp` 里。

智能指针 `boost::shared_ptr` 基本上类似于 `boost::scoped_ptr` 。 关键不同之处在于 `boost::shared_ptr` 不一定要独占一个对象。 它可以和其他 `boost::shared_ptr` 类型的智能指针共享所有权。 在这种情况下，当引用对象的最后一个智能指针销毁后， 对象才会被释放。

因为所有权可以在 `boost::shared_ptr` 之间共享，任何一个共享指针都可以被复制，这跟 `boost::scoped_ptr` 是不同的。 这样就可以在标准容器里存储智能指针了——你不能在标准容器中存储 `std::auto_ptr` ，因为它们在拷贝的时候传递了所有权。

```
#include <boost/shared_ptr.hpp>
#include <vector>

int main()
{
  std::vector<boost::shared_ptr<int> > v;
  v.push_back(boost::shared_ptr<int>(new int(1)));
  v.push_back(boost::shared_ptr<int>(new int(2)));
}
```

- 下载源代码

多亏了有 `boost::shared_ptr`，我们才能像上例中展示的那样，在标准容器中安全的使用动态分配的对象。因为 `boost::shared_ptr` 能够共享它所含对象的所有权，所以保存在容器中的拷贝（包括容器在需要时额外创建的拷贝）都是和原件相同的。如前所述，`std::auto_ptr` 做不到这一点，所以绝对不应该在容器中保存它们。

类似于 `boost::scoped_ptr`，`boost::shared_ptr` 类重载了以下这些操作符：`operator*()`，`operator->()` 和 `operator bool()`。另外还有 `get()` 和 `reset()` 函数来获取和重新初始化所包含的对象的地址。

```
#include <boost/shared_ptr.hpp>

int main()
{
  boost::shared_ptr<int> i1(new int(1));
  boost::shared_ptr<int> i2(i1);
  i1.reset(new int(2));
}
```

- 下载源代码

本例中定义了2个共享指针 `i1` 和 `i2`，它们都引用到同一个 `int` 类型的对象。`i1` 通过 `new` 操作符返回的地址显示的初始化，`i2` 通过 `i1` 拷贝构造而来。`i1` 接着调用 `reset()`，它所包含的整数的地址被重新初始化。不过它之前所包含的对象并没有被释放，因为 `i2` 仍然引用着它。智能指针 `boost::shared_ptr` 记录了有多少个共享指针在引用同一个对象，只有在最后一个共享指针销毁时才会释放这个对象。

默认情况下，`boost::shared_ptr` 使用 `delete` 操作符来销毁所含的对象。然而，具体通过什么方法来销毁，是可以指定的，就像下面的例子里所展示的：

```
#include <boost/shared_ptr.hpp>
#include <windows.h>

int main()
{
  boost::shared_ptr<void> h(OpenProcess(PROCESS_SET_INFORMATION, FA
  SetPriorityClass(h.get(), HIGH_PRIORITY_CLASS);
}
```

- 下载源代码

`boost::shared_ptr` 的构造函数的第二个参数是一个普通函数或者函数对象，该参数用来销毁所含的对象。 在本例中，这个参数是 Windows API 函数 `CloseHandle()` 。 当变量 `h` 超出它的作用域时，调用的是这个函数而不是 `delete` 操作符来销毁所含的对象。 为了避免编译错误，该函数只能带一个 `HANDLE` 类型的参数， `CloseHandle()` 正好符合要求。

该例和本章稍早讲述 RAII 习语时所用的例子的运行是一样的。 然而，本例没有单独定义一个 `windows_handle` 类，而是利用了 `boost::shared_ptr` 的特性，给它的构造函数传递一个方法，这个方法会在共享指针超出它的作用域时自动调用。

## 2.6. 共享数组

共享数组的行为类似于共享指针。 关键不同在于共享数组在析构时，默认使用 `delete[]` 操作符来释放所含的对象。 因为这个操作符只能用于数组对象，共享数组必须通过动态分配的数组的地址来初始化。

共享数组对应的类型是 `boost::shared_array` ，它的定义在 `boost/shared_array.hpp` 里。

```
#include <boost/shared_array.hpp>
#include <iostream>

int main()
{
  boost::shared_array<int> i1(new int[2]);
  boost::shared_array<int> i2(i1);
  i1[0] = 1;
  std::cout << i2[0] << std::endl;
}
```

- 下载源代码

就像共享指针那样，所含对象的所有权可以跟其他共享数组来共享。 这个例子中定义了2个变量 `i1` 和 `i2` ，它们引用到同一个动态分配的数组。 `i1` 通过 `operator[]()` 操作符保存了一个整数1——这个整数可以被 `i2` 引用，比如打印到标准输出。

和本章中所有的智能指针一样， `boost::shared_array` 也同样提供了 `get()` 和 `reset()` 方法。 另外还重载了 `operator bool()` 。

## 2.7. 弱指针

到目前为止介绍的各种智能指针都能在不同的场合下独立使用。 相反，弱指针只有在配合共享指针一起使用时才有意义。 弱指针 `boost::weak_ptr` 的定义在 `boost/weak_ptr.hpp` 里。

```
#include <windows.h>
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
#include <iostream>

DWORD WINAPI reset(LPVOID p)
{
  boost::shared_ptr<int> *sh = static_cast<boost::shared_ptr<int>*>
  sh->reset();
  return 0;
}

DWORD WINAPI print(LPVOID p)
{
  boost::weak_ptr<int> *w = static_cast<boost::weak_ptr<int>*>(p);
  boost::shared_ptr<int> sh = w->lock();
  if (sh)
    std::cout << *sh << std::endl;
  return 0;
}

int main()
{
  boost::shared_ptr<int> sh(new int(99));
  boost::weak_ptr<int> w(sh);
  HANDLE threads[2];
  threads[0] = CreateThread(0, 0, reset, &sh, 0, 0);
  threads[1] = CreateThread(0, 0, print, &w, 0, 0);
  WaitForMultipleObjects(2, threads, TRUE, INFINITE);
}
```

- 下载源代码

`boost::weak_ptr` 必定总是通过 `boost::shared_ptr` 来初始化的。一旦初始化之后，它基本上只提供一个有用的方法: `lock()` 。此方法返回的 `boost::shared_ptr` 与用来初始化弱指针的共享指针共享所有权。 如果这个共享指针不含有任何对象，返回的共享指针也将是空的。

当函数需要一个由共享指针所管理的对象，而这个对象的生存期又不依赖于这个函数时，就可以使用弱指针。 只要程序中还有一个共享指针掌管着这个对象，函数就可以使用该对象。 如果共享指针复位了，就算函数里能得到一个共享指针，对象也不存在了。

上例的 `main()` 函数中，通过 Windows API 创建了2个线程。 于是乎，该例只能在 Windows 平台上编译运行。

第一个线程函数 `reset()` 的参数是一个共享指针的地址。 第二个线程函数 `print()` 的参数是一个弱指针的地址。 这个弱指针是之前通过共享指针初始化的。

一旦程序启动之后， `reset()` 和 `print()` 就都开始执行了。 不过执行顺序是不确定的。 这就导致了一个潜在的问题： `reset()` 线程在销毁对象的时候 `print()` 线程可能正在访问它。

通过调用弱指针的 `lock()` 函数可以解决这个问题：如果对象存在，那么 `lock()` 函数返回的共享指针指向这个合法的对象。否则，返回的共享指针被设置为0，这等价于标准的null指针。

弱指针本身对于对象的生存期没有任何影响。 `lock()` 返回一个共享指针， `print()` 函数就可以安全的访问对象了。 这就保证了——即使另一个线程要释放对象——由于我们有返回的共享指针，对象依然存在。

## 2.8. 介入式指针

大体上，介入式指针的工作方式和共享指针完全一样。 `boost::shared_ptr` 在内部记录着引用到某个对象的共享指针的数量，可是对介入式指针来说，程序员就得自己来做记录。 对于框架对象来说这就特别有用，因为它们记录着自身被引用的次数。

介入式指针 `boost::intrusive_ptr` 定义在 `boost/intrusive_ptr.hpp` 里。

```
#include <boost/intrusive_ptr.hpp>
#include <atlbase.h>
#include <iostream>

void intrusive_ptr_add_ref(IDispatch *p)
{
  p->AddRef();
}

void intrusive_ptr_release(IDispatch *p)
{
  p->Release();
}

void check_windows_folder()
{
  CLSID clsid;
  CLSIDFromProgID(CComBSTR("Scripting.FileSystemObject"), &clsid);
  void *p;
  CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER, __uuidof(IDispat
  boost::intrusive_ptr<IDispatch> disp(static_cast<IDispatch*>(p));
  CComDispatchDriver dd(disp.get());
  CComVariant arg("C:\\Windows");
  CComVariant ret(false);
  dd.Invoke1(CComBSTR("FolderExists"), &arg, &ret);
  std::cout << (ret.boolVal != 0) << std::endl;
}

void main()
{
  CoInitialize(0);
  check_windows_folder();
  CoUninitialize();
}
```

- 下载源代码

上面的例子中使用了 COM（组件对象模型）提供的函数，于是乎只能在 Windows 平台上编译运行。 COM 对象是使用 `boost::intrusive_ptr` 的绝佳范例，因为 COM 对象需要记录当前有多少指针引用着它。 通过调用 `AddRef()` 和 `Release()` 函数，内部的引用计数分别增 1 或者减 1。当引用计数为 0 时，COM 对象自动销毁。

在 `intrusive_ptr_add_ref()` 和 `intrusive_ptr_release()` 内部调用 `AddRef()` 和 `Release()` 这两个函数，来增加或减少相应 COM 对象的引用计数。 这个例子中用到的 COM 对象名为 'FileSystemObject'，在 Windows 上它是默认可用的。通过这个对象可以访问底层的文件系统，比如检查一个给定的目录是否存在。 在上例中，我们检查 `C:\Windows` 目录是否存在。 具体它在内部是怎么实现的，跟 `boost::intrusive_ptr` 的功能无关，完全取决于 COM。 关键点在

于一旦介入式指针 `disp` 离开了它的作用域—— `check_windows_folder()` 函数的末尾，函数 `intrusive_ptr_release()` 将会被自动调用。这将减少 COM 对象 'FileSystemObject' 的内部引用计数到0，于是该对象就销毁了。

## 2.9. 指针容器

在你见过 Boost C++ 库的各种智能指针之后，应该能够编写安全的代码，来使用动态分配的对象和数组。多数时候，这些对象要存储在容器里——如上所述——使用 `boost::shared_ptr` 和 `boost::shared_array` 这就相当简单了。

```
#include <boost/shared_ptr.hpp>
#include <vector>

int main()
{
  std::vector<boost::shared_ptr<int> > v;
  v.push_back(boost::shared_ptr<int>(new int(1)));
  v.push_back(boost::shared_ptr<int>(new int(2)));
}
```

- 下载源代码

上面例子中的代码当然是正确的，智能指针确实可以这样用，然而因为某些原因，实际情况中并不这么用。第一，反复声明 `boost::shared_ptr` 需要更多的输入。其次，将 `boost::shared_ptr` 拷进，拷出，或者在容器内部做拷贝，需要频繁的增加或者减少内部引用计数，这肯定效率不高。由于这些原因，Boost C++ 库提供了 指针容器 专门用来管理动态分配的对象。

```
#include <boost/ptr_container/ptr_vector.hpp>

int main()
{
  boost::ptr_vector<int> v;
  v.push_back(new int(1));
  v.push_back(new int(2));
}
```

- 下载源代码

`boost::ptr_vector` 类的定义在 `boost/ptr_container/ptr_vector.hpp` 里，它跟前一个例子中用 `boost::shared_ptr` 模板参数来初始化的容器具有相同的工作方式。`boost::ptr_vector` 专门用于动态分配的对象，它使用起来更容易也更高效。`boost::ptr_vector` 独占它所包含的对象，因而容器之外的共享指针不能共享所有权，这跟 std::vector<boost::shared_ptr<int> > 相反。

除了 `boost::ptr_vector` 之外，专门用于管理动态分配对象的容器还包括：`boost::ptr_deque`，`boost::ptr_list`，`boost::ptr_set`，`boost::ptr_map`，`boost::ptr_unordered_set` 和 `boost::ptr_unordered_map`。这些容器等价于C++标准里提供的那些。最后两个容器对应于 `std::unordered_set` 和 `std::unordered_map`，它们作为技术报告1的一部分加入 C++ 标准。如果所使用的 C++ 标准实现不支持技术报告1的话，还可以使用 Boost C++ 库里实现的 `boost::unordered_set` 和 `boost::unordered_map`。

## 2.10. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 使用适当的智能指针优化下面的程序：

```
#include &lt;iostream&gt;
#include &lt;cstring&gt;

char *get(const char *s)
{
  int size = std::strlen(s);
  char *text = new char[size + 1];
  std::strncpy(text, s, size + 1);
  return text;
}

void print(char *text)
{
  std::cout &lt;&lt; text &lt;&lt; std::endl;
}

int main(int argc, char *argv[])
{
  if (argc &lt; 2)
  {
    std::cerr &lt;&lt; argv[0] &lt;&lt; " &lt;data&gt;" &lt;&lt
    return 1;
  }

  char *text = get(argv[1]);
  print(text);
  delete[] text;
}
```

   - 下载源代码

2. 优化下面的程序：

```
#include &lt;vector&gt;

template &lt;typename T&gt;
T *create()
{
  return new T;
}

int main()
{
  std::vector&lt;int*&gt; v;
  v.push_back(create&lt;int&gt;());
}
```

- 下载源代码

# 第 3 章 函数对象

## 目录






## 3.1. 概述

本章介绍的是函数对象，可能称为'高阶函数'更为适合。 它实际上是指那些可以被传入到其它函数或是从其它函数返回的一类函数。 在C++中高阶函数是被实现为函数对象的，所以这个标题还是有意义的。

在这整一章中，将会介绍几个用于处理函数对象的 Boost C++ 库。 其中，Boost.Bind 可替换来自C++标准的著名的 `std::bind1st()` 和 `std::bind2nd()` 函数，而 Boost.Function 则提供了一个用于封装函数指针的类。 最后，Boost.Lambda 则引入了一种创建匿名函数的方法。

## 3.2. Boost.Bind

Boost.Bind 是这样的一个库，它简化了由C++标准中的 `std::bind1st()` 和 `std::bind2nd()` 模板函数所提供的一个机制：将这些函数与几乎不限数量的参数一起使用，就可以得到指定签名的函数。 这种情形的一个最好的例子就是在C++标准中定义的多个不同算法。

```
#include <iostream>
#include <vector>
#include <algorithm>

void print(int i)
{
  std::cout << i << std::endl;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(), print);
}
```

- 下载源代码

算法 `std::for_each()` 要求它的第三个参数是一个仅接受正好一个参数的函数或函数对象。 如果 `std::for_each()` 被执行，指定容器中的所有元素 - 在上例中，这些元素的类型为 `int` - 将按顺序被传入至 `print()` 函数。 但是，如果要使用一个具有不同签名的函数的话，事情就复杂了。 例如，如果要传入的是以下函数 `add()` ，它要将一个常数值加至容器中的每个元素上，并显示结果。

```
void add(int i, int j)
{
  std::cout << i + j << std::endl;
}
```

由于 `std::for_each()` 要求的是仅接受一个参数的函数，所以不能直接传入 `add()` 函数。 源代码必须要修改。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

class add
  : public std::binary_function<int, int, void>
{
public:
  void operator()(int i, int j) const
  {
    std::cout << i + j << std::endl;
  }
};

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(), std::bind1st(add(), 10));
}
```

- 下载源代码

以上程序将值10加至容器 `v` 的每个元素之上，并使用标准输出流显示结果。 源代码必须作出大幅的修改，以实现此功能： `add()` 函数已被转换为一个派生自 `std::binary_function` 的函数对象。

Boost.Bind 简化了不同函数之间的绑定。 它只包含一个 `boost::bind()` 模板函数，定义于 `boost/bind.hpp` 中。 使用这个函数，可以如下实现以上例子：

```
#include <boost/bind.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

void add(int i, int j)
{
  std::cout << i + j << std::endl;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(), boost::bind(add, 10, _1));
}
```

- 下载源代码

象 `add()` 这样的函数不再需要为了要用于 `std::for_each()` 而转换为函数对象。 使用 `boost::bind()`，这个函数可以忽略其第一个参数而使用。

因为 `add()` 函数要求两个参数，两个参数都必须传递给 `boost::bind()`。 第一个参数是常数值10，而第二个参数则是一个怪异的 `_1`。

`_1` 被称为占位符(placeholder)，定义于 Boost.Bind。 除了 `_1`，Boost.Bind 还定义了 `_2` 和 `_3`。 通过使用这些占位符， `boost::bind()` 可以变为一元、二元或三元的函数。 对于 `_1`， `boost::bind()` 变成了一个一元函数 - 即只要求一个参数的函数。 这是必需的，因为 `std::for_each()` 正是要求一个一元函数作为其第三个参数。

当这个程序执行时， `std::for_each()` 对容器 `v` 中的第一个元素调用该一元函数。 元素的值通过占位符 `_1` 传入到一元函数中。 这个占位符和常数值被进一步传递到 `add()` 函数。 通过使用这种机制， `std::for_each()` 只看到了由 `boost::bind()` 所定义的一元函数。 而 `boost::bind()` 本身则只是调用了另一个函数，并将常数值或占位符作为参数传入给它。

下面这个例子通过 `boost::bind()` 定义了一个二元函数，用于 `std::sort()` 算法，该算法要求一个二元函数作为其第三个参数。

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::sort(v.begin(), v.end(), boost::bind(compare, _1, _2));
}
```

- 下载源代码

因为使用了两个占位符 `_1` 和 `_2`，所以 `boost::bind()` 定义了一个二元函数。 `std::sort()` 算法以容器 `v` 的两个元素来调用该函数，并根据返回值来对容器进行排序。 基于 `compare()` 函数的定义，容器将被按降序排列。

但是，由于 `compare()` 本身就是一个二元函数，所以使用 `boost::bind()` 确是多余的。

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::sort(v.begin(), v.end(), compare);
}
```

- 下载源代码

不过使用 `boost::bind()` 还是有意义的。例如，如果容器要按升序排列而又不能修改 `compare()` 函数的定义。

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::sort(v.begin(), v.end(), boost::bind(compare, _2, _1));
}
```

- 下载源代码

该例子仅改变了占位符的顺序： `_2` 被作为第一参数传递，而 `_1` 则被作为第二参数传递至 `compare()` ，这样即可改变排序的顺序。

# 3.3. Boost.Ref

本库 Boost.Ref 通常与 Boost.Bind 一起使用，所以我把它们挨着写。 它提供了两个函数 - `boost::ref()` 和 `boost::cref()` - 定义于 `boost/ref.hpp` .

当要用于 `boost::bind()` 的函数带有至少一个引用参数时，Boost.Ref 就很重要了。 由于 `boost::bind()` 会复制它的参数，所以引用必须特别处理。

```
#include <boost/bind.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

void add(int i, int j, std::ostream &os)
{
  os << i + j << std::endl;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(), boost::bind(add, 10, _1, boost
}
```

- 下载源代码

以上例子使用了上一节中的 `add()` 函数。 不过这一次该函数需要一个流对象的引用来打印信息。 因为传给 `boost::bind()` 的参数是以值方式传递的，所以 `std::cout` 不能直接使用，否则该函数会试图创建它的一份拷贝。

通过使用模板函数 `boost::ref()` ，象 `std::cout` 这样的流就可以被以引用方式传递，也就可以成功编译上面这个例子了。

要以引用方式传递常量对象，可以使用模板函数 `boost::cref()` 。

# 3.4. Boost.Function

为了封装函数指针，Boost.Function 提供了一个名为 `boost::function` 的类。它定义于 `boost/function.hpp` ，用法如下：

```
#include <boost/function.hpp>
#include <iostream>
#include <cstdlib>
#include <cstring>

int main()
{
  boost::function<int (const char*)> f = std::atoi;
  std::cout << f("1609") << std::endl;
  f = std::strlen;
  std::cout << f("1609") << std::endl;
}
```

- 下载源代码

`boost::function` 可以定义一个指针，指向具有特定签名的函数。 以上例子定义了一个指针 `f` ，它可以指向某个接受一个类型为 `const char*` 的参数且返回一个类型为 `int` 的值的函数。 定义完成后，匹配此签名的函数均可赋值给这个指针。 这个例程就是先将 `std::atoi()` 赋值给 `f` ，然后再将它重赋值为 `std::strlen()` 。

注意，给定的数据类型并不需要精确匹配：虽然 `std::strlen()` 是以 `std::size_t` 作为返回类型的，但是它也可以被赋值给 `f` 。

因为 `f` 是一个函数指针，所以被赋值的函数可以通过重载的 `operator()()` 操作符来调用。 取决于当前被赋值的是哪一个函数，在以上例子中将调用 `std::atoi()` 或 `std::strlen()` 。

如果 `f` 未赋予一个函数而被调用，则会抛出一个 `boost::bad_function_call` 异常。

```
#include <boost/function.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::function<int (const char*)> f;
    f("");
  }
  catch (boost::bad_function_call &ex)
  {
    std::cout << ex.what() << std::endl;
  }
}
```

- 下载源代码

注意，将值 0 赋给一个 `boost::function` 类型的函数指针，将会释放当前所赋的函数。 释放之后再调用它也会导致 `boost::bad_function_call` 异常被抛出。 要检查一个函数指针是否被赋值某个函数，可以使用 `empty()` 函数或 `operator bool()` 操作符。

通过使用 Boost.Function，类成员函数也可以被赋值给类型为 `boost::function` 的对象。

```
#include <boost/function.hpp>
#include <iostream>

struct world
{
  void hello(std::ostream &os)
  {
    os << "Hello, world!" << std::endl;
  }
};

int main()
{
  boost::function<void (world*, std::ostream&)> f = &world::hello;
  world w;
  f(&w, boost::ref(std::cout));
}
```

- 下载源代码

在调用这样的一个函数时，传入的第一个参数表示了该函数被调用的那个特定对象。 因此，在模板定义中的左括号后的第一个参数必须是该特定类的指针。 接下来的参数才是表示相应的成员函数的签名。

这个程序还使用了来自 Boost.Ref 库的 `boost::ref()`，它提供了一个方便的机制向 Boost.Function 传递引用。

# 3.5. Boost.Lambda

匿名函数 - 又称为 lambda 函数 - 已经在多种编程语言中存在，但 C++ 除外。 不过在 Boost.Lambda 库的帮助下，现在在 C++ 应用中也可以使用它们了。

lambda 函数的目标是令源代码更为紧凑，从而也更容易理解。 以本章第一节中的代码例子为例。

```
#include <iostream>
#include <vector>
#include <algorithm>

void print(int i)
{
  std::cout << i << std::endl;
}

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(), print);
}
```

- 下载源代码

这段程序接受容器 `v` 中的元素并使用 `print()` 函数将它们写出到标准输出流。 由于 `print()` 只是写出一个简单的 `int` ，所以该函数的实现相当简单。 严格来说，它是如此地简单，以致于如果可以在 `std::for_each()` 算法里面直接定义它的话，会更为方便； 从而省去增加一个函数的需要。 另外一个好处是代码更为紧凑，使得算法与负责数据输出的函数不是局部性分离的。 Boost.Lambda 正好使之成为现实。

```
#include <boost/lambda/lambda.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(), std::cout << boost::lambda::_1
}
```

- 下载源代码

Boost.Lambda 提供了几个结构来定义匿名函数。 代码就被置于执行的地方，从而省去将它包装为一个函数再进行相应的函数调用的这些开销。 与原来的例子一样，这个程序将容器 `v` 的所有元素写出至标准输出流。

与 Boost.Bind 相类似，Boost.Lambda 也定义了三个占位符，名为 `_1`，`_2` 和 `_3` 。 但与 Boost.Bind 不同的是，这些占位符是定义在单独的名字空间的。 因此，该例中的第一个占位符是通过 `boost::lambda::_1` 来引用的。 为了满足编译器的要求，必须包含相应的头文件 `boost/lambda/lambda.hpp` 。

虽然代码的位置位于 `std::for_each()` 的第三个参数处，看起来很怪异，但 Boost.Lambda 可以写出正常的 C++ 代码。 通过使用占位符，容器 `v` 的元素可以通过 `<<` 传给 `std::cout` 以将它们写出到标准输出流。

虽然 Boost.Lambda 非常强大，但也有一些缺点。 要在以上例子中插入换行的话，必须用 "\n" 来替代 `std::endl` 才能成功编译。 因为一元 `std::endl` 模板函数所要求的类型不同于 lambda 函数 `std::cout << boost::lambda::_1` 的函数，所以在此不能使用它。

下一个版本的 C++ 标准很可能会将 lambda 函数作为 C++ 语言本身的组成部分加入，从而消除对单独的库的需要。 但是在下一个版本到来并被不同的编译器厂商所采用可能还需要好几年。 在此之前，Boost.Lambda 被证明是一个完美的替代品，从以下例子可以看出，这个例子只将大于1的元素写出到标准输出流。

```
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/if.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);

  std::for_each(v.begin(), v.end(),
    boost::lambda::if_then(boost::lambda::_1 > 1,
    std::cout << boost::lambda::_1 << "\n"));
}
```

- 下载源代码

头文件 `boost/lambda/if.hpp` 定义了几个结构，允许在 lambda 函数内部使用 `if` 语句。 最基本的结构是 `boost::lambda::if_then()` 模板函数，它要求两个参数：第一个参数对条件求值 - 如果为真，则执行第二个参数。 如例中所示，每个参数本身都可以是 lambda 函数。

除了 `boost::lambda::if_then()`, Boost.Lambda 还提供了 `boost::lambda::if_then_else()` 和 `boost::lambda::if_then_else_return()` 模板函数 - 它们都要求三个参数。 另外还提供了用于实现循环、转型操作符，甚至是 `throw` - 允许 lambda 函数抛出异常 - 的模板函数。

虽然可以用这些模板函数在 C++ 中构造出复杂的 lambda 函数，但是你必须要考虑其它方面，如可读性和可维护性。 因为别人需要学习并理解额外的函数，如用 `boost::lambda::if_then()` 来替代已知的 C++ 关键字 `if` 和 `else` ，lambda 函数的好处通常随着它的复杂性而降低。 多数情况下，更为合理的方法是用熟悉的 C++ 结构定义一个单独的函数。

## 3.6. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 简化以下程序，将函数对象 `divide_by` 转换为一个函数，并将 `for` 循环替换为用一个标准的 C++ 算法来输出数据：

```
#include &lt;algorithm&gt;
#include &lt;functional&gt;
#include &lt;vector&gt;
#include &lt;iostream&gt;

class divide_by
  : public std::binary_function&lt;int, int, int&gt;
{
public:
  int operator()(int n, int div) const
  {
    return n / div;
  }
};

int main()
{
  std::vector&lt;int&gt; numbers;
  numbers.push_back(10);
  numbers.push_back(20);
  numbers.push_back(30);

  std::transform(numbers.begin(), numbers.end(), numbers.begin(

  for (std::vector&lt;int&gt;::iterator it = numbers.begin(); i
    std::cout &lt;&lt; *it &lt;&lt; std::endl;
}
```

   - 下载源代码

2. 简化以下程序，将两个 `for` 循环都替换为标准的 C++ 算法：

```
#include &lt;string&gt;
#include &lt;vector&gt;
#include &lt;iostream&gt;

int main()
{
  std::vector&lt;std::string&gt; strings;
  strings.push_back("Boost");
  strings.push_back("C++");
  strings.push_back("Libraries");

  std::vector&lt;int&gt; sizes;

  for (std::vector&lt;std::string&gt;::iterator it = strings.be
    sizes.push_back(it-&gt;size());

  for (std::vector&lt;int&gt;::iterator it = sizes.begin(); it
    std::cout &lt;&lt; *it &lt;&lt; std::endl;
}
```

- 下载源代码

3. 简化以下程序，修改变量 `processors` 的类型，并将 `for` 循环替换为标准的 C++ 算法：

```
#include &lt;vector&gt;
#include &lt;iostream&gt;
#include &lt;cstdlib&gt;
#include &lt;cstring&gt;

int main()
{
  std::vector&lt;int(*)(const char*)&gt; processors;
  processors.push_back(std::atoi);
  processors.push_back(reinterpret_cast&lt;int(*)(const char*)&

  const char data[] = "1.23";

  for (std::vector&lt;int(*)(const char*)&gt;::iterator it = pr
    std::cout &lt;&lt; (*it)(data) &lt;&lt; std::endl;
}
```

- 下载源代码

# 第 4 章 事件处理

## 目录






## 4.1. 概述

很多开发者在听到术语'事件处理'时就会想到GUI：点击一下某个按钮，相关联的功能就会被执行。 点击本身就是事件，而功能就是相对应的事件处理器。

这一模式的使用当然不仅限于GUI。 一般情况下，任意对象都可以调用基于特定事件的专门函数。 本章所介绍的 Boost.Signals 库提供了一个简单的方法在 C++ 中应用这一模式。

严格来说，Boost.Function 库也可以用于事件处理。 不过，Boost.Function 和 Boost.Signals 之间的一个主要区别在于，Boost.Signals 能够将一个以上的事件处理器关联至单个事件。 因此，Boost.Signals 可以更好地支持事件驱动的开发，当需要进行事件处理时，应作为第一选择。

## 4.2. 信号 Signals

虽然这个库的名字乍一看好象有点误导，但实际上并非如此。 Boost.Signals 所实现的模式被命名为 '信号至插槽' (signal to slot)，它基于以下概念：当对应的信号被发出时，相关联的插槽即被执行。 原则上，你可以把单词 '信号' 和 '插槽' 分别替换为 '事件' 和 '事件处理器'。 不过，由于信号可以在任意给定的时间发出，所以这一概念放弃了 '事件' 的名字。

因此，Boost.Signals 没有提供任何类似于 '事件' 的类。 相反，它提供了一个名为 `boost::signal` 的类，定义于 `boost/signal.hpp`. 实际上，这个头文件是唯一一个需要知道的，因为它会自动包含其它相关的头文件。

Boost.Signals 定义了其它一些类，位于 boost::signals 名字空间中。 由于 `boost::signal` 是最常被用到的类，所以它是位于名字空间 boost 中的。

```
#include <boost/signal.hpp>
#include <iostream>

void func()
{
  std::cout << "Hello, world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  s.connect(func);
  s();
}
```

- 下载源代码

`boost::signal` 实际上被实现为一个模板函数，具有被用作为事件处理器的函数的签名，该签名也是它的模板参数。 在这个例子中，只有签名为 `void ()` 的函数可以被成功关联至信号 `s` 。

函数 `func()` 被通过 `connect()` 方法关联至信号 `s` 。 由于 `func()` 符合所要求的 `void ()` 签名，所以该关联成功建立。因此当信号 `s` 被触发时， `func()` 将被调用。

信号是通过调用 `s` 来触发的，就象普通的函数调用那样。 这个函数的签名对应于作为模板参数传入的签名：因为 `void ()` 不要求任何参数，所以括号内是空的。

调用 `s` 会引发一个触发器，进而执行相应的 `func()` 函数 - 之前用 `connect()` 关联了的。

同一例子也可以用 Boost.Function 来实现。

```
#include <boost/function.hpp>
#include <iostream>

void func()
{
  std::cout << "Hello, world!" << std::endl;
}

int main()
{
  boost::function<void ()> f;
  f = func;
  f();
}
```

- 下载源代码

和前一个例子相类似， `func()` 被关联至 `f` 。 当 `f` 被调用时，就会相应地执行 `func()` 。 Boost.Function 仅限于这种情形下适用，而 Boost.Signals 则提供了多得多的方式，如关联多个函数至单个特定信号，示例如下。

```
#include <boost/signal.hpp>
#include <iostream>

void func1()
{
  std::cout << "Hello" << std::flush;
}

void func2()
{
  std::cout << ", world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  s.connect(func1);
  s.connect(func2);
  s();
}
```

- 下载源代码

`boost::signal` 可以通过反复调用 `connect()` 方法来把多个函数赋值给单个特定信号。 当该信号被触发时，这些函数被按照之前用 `connect()` 进行关联时的顺序来执行。

另外，执行的顺序也可通过 `connect()` 方法的另一个重载版本来明确指定，该重载版本要求以一个 `int` 类型的值作为额外的参数。

```
#include <boost/signal.hpp>
#include <iostream>

void func1()
{
  std::cout << "Hello" << std::flush;
}

void func2()
{
  std::cout << ", world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  s.connect(1, func2);
  s.connect(0, func1);
  s();
}
```

- 下载源代码

和前一个例子一样，`func1()` 在 `func2()` 之前执行。

要释放某个函数与给定信号的关联，可以用 `disconnect()` 方法。

```
#include <boost/signal.hpp>
#include <iostream>

void func1()
{
  std::cout << "Hello" << std::endl;
}

void func2()
{
  std::cout << ", world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  s.connect(func1);
  s.connect(func2);
  s.disconnect(func2);
  s();
}
```

- 下载源代码

这个例子仅输出 `Hello` ，因为与 `func2()` 的关联在触发信号之前已经被释放。

除了 `connect()` 和 `disconnect()` 以外， `boost::signal` 还提供了几个方法。

```
#include <boost/signal.hpp>
#include <iostream>

void func1()
{
  std::cout << "Hello" << std::flush;
}

void func2()
{
  std::cout << ", world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  s.connect(func1);
  s.connect(func2);
  std::cout << s.num_slots() << std::endl;
  if (!s.empty())
    s();
  s.disconnect_all_slots();
}
```

- 下载源代码

`num_slots()` 返回已关联函数的数量。如果没有函数被关联，则 `num_slots()` 返回0。 在这种特定情况下，可以用 `empty()` 方法来替代。 `disconnect_all_slots()` 方法所做的实际上正是它的名字所表达的：释放所有已有的关联。

看完了函数如何被关联至信号，以及弄明白了信号被触发时会发生什么事之后，还有一个问题：这些函数的返回值去了哪里？ 以下例子回答了这个问题。

```
#include <boost/signal.hpp>
#include <iostream>

int func1()
{
  return 1;
}

int func2()
{
  return 2;
}

int main()
{
  boost::signal<int ()> s;
  s.connect(func1);
  s.connect(func2);
  std::cout << s() << std::endl;
}
```

- 下载源代码

`func1()` 和 `func2()` 都具有 `int` 类型的返回值。 `s` 将处理两个返回值，并将它们都写出至标准输出流。 那么，到底会发生什么呢？

以上例子实际上会把 `2` 写出至标准输出流。 两个返回值都被 `s` 正确接收，但除了最后一个值，其它值都会被忽略。 缺省情况下，所有被关联函数中，实际上只有最后一个返回值被返回。

你可以定制一个信号，令每个返回值都被相应地处理。 为此，要把一个称为合成器(combiner)的东西作为第二个参数传递给 `boost::signal` 。

```
#include <boost/signal.hpp>
#include <iostream>
#include <algorithm>

int func1()
{
  return 1;
}

int func2()
{
  return 2;
}

template <typename T>
struct min_element
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    return *std::min_element(first, last);
  }
};

int main()
{
  boost::signal<int (), min_element<int> > s;
  s.connect(func1);
  s.connect(func2);
  std::cout << s() << std::endl;
}
```

- 下载源代码

合成器是一个重载了 `operator()()` 操作符的类。这个操作符会被自动调用，传入两个迭代器，指向某个特定信号的所有返回值。 以上例子使用了标准 C++ 算法 `std::min_element()` 来确定并返回最小的值。

不幸的是，我们不可能把象 `std::min_element()` 这样的一个算法直接传给 `boost::signal` 作为一个模板参数。 `boost::signal` 要求这个合成器定义一个名为 `result_type` 的类型，用于说明 `operator()()` 操作符返回值的类型。 由于在标准 C++ 算法中缺少这个类型，所以在编译时会产生一个相应的错误。

除了对返回值进行分析以外，合成器也可以保存它们。

```
#include <boost/signal.hpp>
#include <iostream>
#include <vector>
#include <algorithm>

int func1()
{
  return 1;
}

int func2()
{
  return 2;
}

template <typename T>
struct min_element
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    return T(first, last);
  }
};

int main()
{
  boost::signal<int (), min_element<std::vector<int> > > s;
  s.connect(func1);
  s.connect(func2);
  std::vector<int> v = s();
  std::cout << *std::min_element(v.begin(), v.end()) << std::endl;
}
```

- 下载源代码

这个例子把所有返回值保存在一个 vector 中，再由 `s()` 返回。

## 4.3. 连接 Connections

函数可以通过由 `boost::signal` 所提供的 `connect()` 和 `disconnect()` 方法的帮助来进行管理。 由于 `connect()` 会返回一个类型为 `boost::signals::connection` 的值，它们可以通过其它方法来管理。

```
#include <boost/signal.hpp>
#include <iostream>

void func()
{
  std::cout << "Hello, world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  boost::signals::connection c = s.connect(func);
  s();
  c.disconnect();
}
```

- 下载源代码

`boost::signal` 的 `disconnect()` 方法需要传入一个函数指针，而直接调用 `boost::signals::connection` 对象上的 `disconnect()` 方法则略去该参数。

除了 `disconnect()` 方法之外， `boost::signals::connection` 还提供了其它方法，如 `block()` 和 `unblock()` 。

```
#include <boost/signal.hpp>
#include <iostream>

void func()
{
  std::cout << "Hello, world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  boost::signals::connection c = s.connect(func);
  c.block();
  s();
  c.unblock();
  s();
}
```

- 下载源代码

以上程序只会执行一次 `func()` 。 虽然信号 `s` 被触发了两次，但是在第一次触发时 `func()` 不会被调用，因为连接 `c` 实际上已经被 `block()` 调用所阻塞。由于在第二次触发之前调用了 `unblock()` ，所以之后 `func()` 被正确地执行。

除了 `boost::signals::connection` 以外，还有一个名为 `boost::signals::scoped_connection` 的类，它会在析构时自动释放连接。

```
#include <boost/signal.hpp>
#include <iostream>

void func()
{
  std::cout << "Hello, world!" << std::endl;
}

int main()
{
  boost::signal<void ()> s;
  {
    boost::signals::scoped_connection c = s.connect(func);
  }
  s();
}
```

- 下载源代码

因为连接对象 `c` 在信号触发之前被销毁，所以 `func()` 不会被调用。

`boost::signals::scoped_connection` 实际上是派生自 `boost::signals::connection` 的，所以它提供了相同的方法。它们之间的区别仅在于，在析构 `boost::signals::scoped_connection` 时，连接会自动释放。

虽然 `boost::signals::scoped_connection` 的确令自动释放连接更为容易，但是该类型的对象仍需要管理。 如果在其它情形下连接也可以被自动释放，而且不需要管理这些对象的话，就更好了。

```
#include <boost/signal.hpp>
#include <boost/bind.hpp>
#include <iostream>
#include <memory>

class world
{
  public:
    void hello() const
    {
      std::cout << "Hello, world!" << std::endl;
    }
};

int main()
{
  boost::signal<void ()> s;
  {
    std::auto_ptr<world> w(new world());
    s.connect(boost::bind(&world::hello, w.get()));
  }
  std::cout << s.num_slots() << std::endl;
  s();
}
```

- 下载源代码

以上程序使用 Boost.Bind 将一个对象的方法关联至一个信号。 在信号触发之前，这个对象就被销毁了，这会产生问题。 我们不传递实际的对象 `w` ，而只传递一个指针给 `boost::bind()` 。 在 `s()` 被实际调用的时候，该指针所引向的对象已不再存在。

可以如下修改这个程序，使得一旦对象 `w` 被销毁，连接就会自动释放。

```
#include <boost/signal.hpp>
#include <boost/bind.hpp>
#include <iostream>
#include <memory>

class world :
  public boost::signals::trackable
{
  public:
    void hello() const
    {
      std::cout << "Hello, world!" << std::endl;
    }
};

int main()
{
  boost::signal<void ()> s;
  {
    std::auto_ptr<world> w(new world());
    s.connect(boost::bind(&world::hello, w.get()));
  }
  std::cout << s.num_slots() << std::endl;
  s();
}
```

- 下载源代码

如果现在再执行, `num_slots()` 会返回 `0` 以确保不会试图调用已销毁对象之上的方法。 仅需的修改是让 `world` 类继承自 `boost::signals::trackable` 。 当使用对象的指针而不是对象的副本来关联函数至信号时, `boost::signals::trackable` 可以显著简化连接的管理。

## 4.4. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 编写一个程序，定义一个名为 `button` 的类，表示GUI中的一个可点击按钮。 为该类加入两个方法 `add_handler()` 和 `remove_handler()` ，它们均要求一个函数名作为参数。 如果 `click()` 方法被调用，已登记的函数将被按顺序执行。

   如下测试你的代码，创建一个 `button` 类的实例，从事件处理器内部向标准输出流写出一个信息。 调用 `click()` 函数模拟用鼠标点击该按钮。

# 第 5 章 字符串处理

## 目录







## 5.1. 前言

在标准 C++ 中，用于处理字符串的是 `std::string` 类，它提供很多字符串操作，包括查找指定字符或子串的函数。 尽管 `std::string` 囊括了百余函数，是标准 C++ 中最为臃肿的类之一，然而却并不能满足很多开发者在日常工作中的需要。 例如， Java 和 .Net 提供了可以将字符串转换到大写字母的函数，而 `std::string` 就没有相应的功能。 Boost C++ 库试图弥补这一缺憾。

## 5.2. 区域设置

在进入正题之前，有必要先审视下区域设置的问题，本章中提到的很多函数都需要一个附加的区域设置参数。

区域设置在标准 C++ 中封装了文化习俗相关的内容，包括货币符号，日期时间格式， 分隔整数部分与分数部分的符号（基数符）以及多于三个数字时的分隔符（千位符）。

在字符串处理方面，区域设置和特定文化中对字符次序以及特殊字符的描述有关。 例如，字母表中是否含有变异元音字母以及其在字母表中的位置都由语言文化决定。

如果一个函数用于将字符串转换为大写形式，那么其实施步骤取决于具体的区域设置。 在德语中，字母 'ä' 显然要转换为 'Ä'， 然而在其他语言中并不一定。

使用类 `std::string` 时区域设置可以忽略， 因为它的函数均不依赖于特定语言。 然而在本章中为了使用 Boost C++ 库， 区域设置的知识是必不可少的。

C++ 标准中在 `locale` 文件中定义了类 `std::locale` 。 每个 C++ 程序自动拥有一个此类的实例， 即不能直接访问的全局区域设置。 如果要访问它，需要使用默认构造函数构造类 `std::locale` 的对象，并使用与全局区域设置相同的属性

初始化。

```
#include <locale>
#include <iostream>

int main()
{
  std::locale loc;
  std::cout << loc.name() << std::endl;
}
```

- 下载源代码

以上程序在标准输出流输出 `C` ，这是基本区域设置的名称，它包括了 C 语言编写的程序中默认使用的描述。

这也是每个 C++ 应用的默认全局区域设置，它包括了美式文化中使用的描述。 例如，货币符号使用美元符号，基字符为英文句号，日期中的月份用英语书写。

全局区域设置可以使用类 `std::locale` 中的静态函数 `global()` 改变。

```
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::locale loc;
  std::cout << loc.name() << std::endl;
}
```

- 下载源代码

静态函数 `global()` 接受一个类型为 `std::locale` 的对象作为其唯一的参数。此类的另一个版本的构造函数接受类型为 `const char*` 的字符串，可以为一个特别的文化创建区域设置对象。 然而，除了 C 区域设置相应地命名为 "C" 之外，其他区域设置的名字并没有标准化，所以这依赖于接受区域设置名字的 C++ 标准库。 在使用 Visual Studio 2008 的情况下，语言字符串文档 指出， 可以使用语言字符串 "German" 选择定义为德国文化。

执行程序，会输出 `German_Germany.1252` 。 指定语言字符串为 "German" 等于选择了德国文化作为主要语言和子语言，这里选择了字符映射 1252。

如果想指定与德国文化不同的子语言设置，例如瑞士语，需要使用不同的语言字符串。

```
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German_Switzerland"));
  std::locale loc;
  std::cout << loc.name() << std::endl;
}
```

- 下载源代码

现在程序会输出 `German_Switzerland.1252` 。

在初步理解了区域设置以及如何更改全局设置后， 下面的例子说明了区域设置如何影响字符串操作。

```
#include <locale>
#include <iostream>
#include <cstring>

int main()
{
  std::cout << std::strcoll("ä", "z") << std::endl;
  std::locale::global(std::locale("German"));
  std::cout << std::strcoll("ä", "z") << std::endl;
}
```

- 下载源代码

本例使用了定义在文件 `cstring` 中的函数 `std::strcoll()` ， 这个函数用于按照字典顺序比较第一个字符串是否小于第二个。 换言之， 它会判断这两个字符串中哪一个在字典中靠前。

执行程序， 得到结果为 `1` 和 `-1` 。 虽然函数的参数是一样的， 却得到了不同的结果。 原因很简单， 在第一次调用函数 `std::strcoll()` 时， 使用了全局 C 区域设置 ； 而在第二次调用时， 全局区域设置更改为德国文化。 从输出中可以看出， 在这两种区域设置中， 字符 'ä' 和 'z' 的次序是不同的。

很多 C 函数以及 C++ 流都与区域设置有关。 尽管类 `std::string` 中的函数是与区域设置独立工作的， 但是以下各节中提到的函数并不是这样。 所以， 在本章中还会多次提到区域设置的相关内容。

## 5.3. 字符串算法库 Boost.StringAlgorithms

Boost C++ 字符串算法库 Boost.StringAlgorithms 提供了很多字符串操作函数。 字符串的类型可以是 `std::string` ， `std::wstring` 或任何其他模板类 `std::basic_string` 的实例。

这些函数分类别在不同的头文件定义。 例如，大小写转换函数定义在文件 `boost/algorithm/string/case_conv.hpp` 中。 因为 Boost.StringAlgorithms 类中包括超过20个类别和相同数目的头文件， 为了方便起见， 头文件 `boost/algorithm/string.hpp` 包括了所有其他的头文件。 后面所有例子都会使用这个头文件。

正如上节提到的那样， Boost.StringAlgorithms 库中许多函数 都可以接受一个类型为 `std::locale` 的对象作为附加参数。 而此参数是可选的，如果不设置将使用默认全局区域设置。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>
#include <clocale>

int main()
{
  std::setlocale(LC_ALL, "German");
  std::string s = "Boris Schäling";
  std::cout << boost::algorithm::to_upper_copy(s) << std::endl;
  std::cout << boost::algorithm::to_upper_copy(s, std::locale("Gern
}
```

- 下载源代码

函数 `boost::algorithm::to_upper_copy()` 用于 转换一个字符串为大写形式，自然也有提供相反功能的函数 —— `boost::algorithm::to_lower_copy()` 把字符串转换为小写形式。 这两个函数都返回转换过的字符串作为结果。 如果作为参数传入的字符串自身需要被转换为大（小）写形式，可以使用函数 `boost::algorithm::to_upper()` 或 `boost::algorithm::to_lower ()` 。

上面的例子使用函数 `boost::algorithm::to_upper_copy()` 把字符串 "Boris Schäling" 转换为大写形式。 第一次调用时使用的是默认全局区域设置， 第二次调用时则明确将区域设置为德国文化。

显然后者的转换是正确的， 因为小写字母 'ä' 对应的大写形式 'Ä' 是存在的。 而在 C 区域设置中， 'ä' 是一个未知字符所以不能转换。 为了能得到正确结果，必须明确传递正确的区域设置参数或者在调用 `boost::algorithm::to_upper_copy()` 之前改变全局区域设置。

可以注意到，程序使用了定义在头文件 `clocale` 中的函数 `std::setlocale()` 为 C 函数进行区域设置， 因为 `std::cout` 使用 C 函数在屏幕上显示信息。 在设置了正确的区域后，才可以正确显示 'ä' 和 'Ä' 等元音字母。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  std::cout << boost::algorithm::to_upper_copy(s) << std::endl;
  std::cout << boost::algorithm::to_upper_copy(s, std::locale("Gern
}
```

- 下载源代码

上述程序将全局区域设置设为德国文化，这使得对函数 `boost::algorithm::to_upper_copy()` 的调用 可以将 'ä' 转换为 'Ä' 。

注意到本例并没有调用函数 `std::setlocale()` 。 使用函数 `std::locale::global()` 设置全局区域设置后， 也自动进行了 C 区域设置。 实际上，C++ 程序几乎总是使用函数 `std::locale::global()` 进行全局区域设置， 而不是像前面的例子那样使用函数 `std::setlocale()` 。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  std::cout << boost::algorithm::erase_first_copy(s, "i") << std::e
  std::cout << boost::algorithm::erase_nth_copy(s, "i", 0) << std:
  std::cout << boost::algorithm::erase_last_copy(s, "i") << std::er
  std::cout << boost::algorithm::erase_all_copy(s, "i") << std::enc
  std::cout << boost::algorithm::erase_head_copy(s, 5) << std::endl
  std::cout << boost::algorithm::erase_tail_copy(s, 8) << std::endl
}
```

- 下载源代码

Boost.StringAlgorithms 库提供了几个从字符串中删除单独字母的函数， 可以明确指定在哪里删除，如何删除。例如，可以使用函数 `boost::algorithm::erase_all_copy()` 从整个字符串中 删除特定的某个字符。 如果只在此字符首次出现时删除，可以使用函数

boost::algorithm::erase_first_copy() 。 如果要在字符串头部或尾部删除若干字符，可以使用函数 boost::algorithm::erase_head_copy() 和 boost::algorithm::erase_tail_copy() 。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  boost::iterator_range<std::string::iterator> r = boost::algorithm
  std::cout << r << std::endl;
  r = boost::algorithm::find_first(s, "xyz");
  std::cout << r << std::endl;
}
```

- 下载源代码

以下各个不同函数 boost::algorithm::find_first() 、 boost::algorithm::find_last() 、 boost::algorithm::find_nth() 、 boost::algorithm::find_head() 以及 boost::algorithm::find_tail() 可以用于在字符串中查找子串。

所有这些函数的共同点是均返回类型为 boost::iterator_range 类 的一对迭代器。 此类起源于 Boost C++ 的 Boost.Range 库， 它在迭代器的概念上定义了“范围”。 因为操作符 << 由 boost::iterator_range 类重载而来， 单个搜索算法的结果可以直接写入标准输出流。 以上程序将 Boris 作为第一个结果输出而第二个结果为空字符串。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>
#include <vector>

int main()
{
  std::locale::global(std::locale("German"));
  std::vector<std::string> v;
  v.push_back("Boris");
  v.push_back("Schäling");
  std::cout << boost::algorithm::join(v, " ") << std::endl;
}
```

- 下载源代码

函数 `boost::algorithm::join()` 接受一个字符串的容器 作为第一个参数， 根据第二个参数将这些字符串连接起来。 相应地这个例子会输出 `Boris Schäling` 。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  std::cout << boost::algorithm::replace_first_copy(s, "B", "D") <<
  std::cout << boost::algorithm::replace_nth_copy(s, "B", 0, "D") <
  std::cout << boost::algorithm::replace_last_copy(s, "B", "D") <<
  std::cout << boost::algorithm::replace_all_copy(s, "B", "D") << s
  std::cout << boost::algorithm::replace_head_copy(s, 5, "Doris") <
  std::cout << boost::algorithm::replace_tail_copy(s, 8, "Becker")
}
```

- 下载源代码

Boost.StringAlgorithms 库不但提供了查找子串或删除字母的函数， 而且提供了使用字符串替代子串的函数，包括 `boost::algorithm::replace_first_copy()` , `boost::algorithm::replace_nth_copy()` , `boost::algorithm::replace_last_copy()` , `boost::algorithm::replace_all_copy()` , `boost::algorithm::replace_head_copy()` 以及 `boost::algorithm::replace_tail_copy()` 等等。 它们的使用方法同查找和删除函数是差不多一样的，所不同的是还需要 一个替代字符串作为附加参数。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "\t Boris Schäling \t";
  std::cout << "." << boost::algorithm::trim_left_copy(s) << "." <<
  std::cout << "." <<boost::algorithm::trim_right_copy(s) << "." <<
  std::cout << "." <<boost::algorithm::trim_copy(s) << "." << std:
}
```

- 下载源代码

可以使用修剪函数 `boost::algorithm::trim_left_copy()` , `boost::algorithm::trim_right_copy()` 以及 `boost::algorithm::trim_copy()` 等自动去除字符串中的空格或者字符串的结束符。 什么字符是空格取决于全局区域设置。

Boost.StringAlgorithms 库的函数可以接受一个附加的谓词参数，以决定函数作用于字符串的哪些字符。 谓词版本的修剪函数相应地被命名为 `boost::algorithm::trim_left_copy_if()` , `boost::algorithm::trim_right_copy_if()` 和 `boost::algorithm::trim_copy_if()` 。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "--Boris Schäling--";
  std::cout << "." << boost::algorithm::trim_left_copy_if(s, boost
  std::cout << "." <<boost::algorithm::trim_right_copy_if(s, boost
  std::cout << "." <<boost::algorithm::trim_copy_if(s, boost::algo
}
```

- 下载源代码

以上程序调用了一个辅助函数 `boost::algorithm::is_any_of()` ， 它用于生成谓词以验证作为参数传入的字符是否在给定的字符串中存在。使用函数 `boost::algorithm::is_any_of` 后，正如例子中做的那样，修剪字符串的字符被指定为

Boost.StringAlgorithms 类也提供了众多返回通用谓词的辅助函数。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "123456789Boris Schäling123456789";
  std::cout << "." << boost::algorithm::trim_left_copy_if(s, boost
  std::cout << "." <<boost::algorithm::trim_right_copy_if(s, boost
  std::cout << "." <<boost::algorithm::trim_copy_if(s, boost::algo
}
```

- 下载源代码

函数 `boost::algorithm::is_digit()` 返回的谓词在字符为数字时返回布尔值 `true` 。 检查字符是否为大写或小写的辅助函数分别是 `boost::algorithm::is_upper()` 和 `boost::algorithm::is_lower()` 。 所有这些函数都默认使用全局区域设置，除非在参数中指定其他区域设置。

除了检验单独字符的谓词之外， Boost.StringAlgorithms 库还提供了处理字符串的函数。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  std::cout << boost::algorithm::starts_with(s, "Boris") << std::er
  std::cout << boost::algorithm::ends_with(s, "Schäling") << std::e
  std::cout << boost::algorithm::contains(s, "is") << std::endl;
  std::cout << boost::algorithm::lexicographical_compare(s, "Boris'
}
```

- 下载源代码

函数 `boost::algorithm::starts_with()` 、 `boost::algorithm::ends_with()` 、 `boost::algorithm::contains()` 和 `boost::algorithm::lexicographical_compare()` 均可以比较两个字符串。

以下介绍一个字符串切割函数。

```
#include <boost/algorithm/string.hpp>
#include <locale>
#include <iostream>
#include <vector>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  std::vector<std::string> v;
  boost::algorithm::split(v, s, boost::algorithm::is_space());
  std::cout << v.size() << std::endl;
}
```

- 下载源代码

在给定分界符后，使用函数 `boost::algorithm::split()` 可以将一个字符串拆分为一个字符串容器。 它需要给定一个谓词作为第三个参数以判断应该在字符串的哪个位置分割。 这个例子使用了辅助函数 `boost::algorithm::is_space()` 创建一个谓词，在每个空格字符处分割字符串。

本节中许多函数都有忽略字符串大小写的版本， 这些版本一般都有与原函数相似的名称，所相差的只是以 'i'.开头。 例如，与函数 `boost::algorithm::erase_all_copy()` 相对应的是函数 `boost::algorithm::ierase_all_copy()` 。

最后， 值得注意的是类 Boost.StringAlgorithms 中许多函数都支持正则表达式。 以下程序使用函数 `boost::algorithm::find_regex()` 搜索正则表达式。

```
#include <boost/algorithm/string.hpp>
#include <boost/algorithm/string/regex.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  boost::iterator_range<std::string::iterator> r = boost::algorithr
  std::cout << r << std::endl;
}
```

- 下载源代码

为了使用正则表达式，此程序使用了Boost C++ 库中的 `boost::regex` ， 这将在下一节介绍。

## 5.4. 正则表达式库 Boost.Regex

Boost C++ 的正则表达式库 Boost.Regex 可以应用正则表达式于 C++ 。 正则表达式大大减轻了搜索特定模式字符串的负担，在很多语言中都是强大的功能。 虽然现在 C++ 仍然需要以 Boost C++ 库的形式提供这一功能，但是在将来正则表达式将进入 C++ 标准库。 Boost.Regex 库有望包括在下一版的 C++ 标准中。

Boost.Regex 库中两个最重要的类是 `boost::regex` 和 `boost::smatch`， 它们都在 `boost/regex.hpp` 文件中定义。 前者用于定义一个正则表达式，而后者可以保存搜索结果。

以下将要介绍 Boost.Regex 库中提供的三个搜索正则表达式的函数。

```
#include <boost/regex.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  boost::regex expr("\\w+\\s\\w+");
  std::cout << boost::regex_match(s, expr) << std::endl;
}
```

- 下载源代码

函数 `boost::regex_match()` 用于字符串与正则表达式的比较。 在整个字符串匹配正则表达式时其返回值为 `true` 。

函数 `boost::regex_search()` 可用于在字符串中搜索正则表达式。

```
#include <boost/regex.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  boost::regex expr("(\\w+)\\s(\\w+)");
  boost::smatch what;
  if (boost::regex_search(s, what, expr))
  {
    std::cout << what[0] << std::endl;
    std::cout << what[1] << " " << what[2] << std::endl;
  }
}
```

- 下载源代码

函数 `boost::regex_search()` 可以接受一个类型为 `boost::smatch` 的引用的参数用于储存结果。 函数 `boost::regex_search()` 只用于分类的搜索， 本例实际上返回了两个结果， 它们是基于正则表达式的分组。

存储结果的类 `boost::smatch` 事实上是持有类型为 `boost::sub_match` 的元素的容器， 可以通过与类 `std::vector` 相似的界面访问。 例如， 元素可以通过操作符 `operator[]()` 访问。

另一方面， 类 `boost::sub_match` 将迭代器保存在对应于正则表达式分组的位置。 因为它继承自类 `std::pair` ， 迭代器引用的子串可以使用 `first` 和 `second` 访问。如果像上面的例子那样， 只把子串写入标准输出流， 那么通过重

载操作符 << 就可以直接做到这一点，那么并不需要访问迭代器。

请注意结果保存在迭代器中而 `boost::sub_match` 类并不复制它们，这说明它们只是在被迭代器引用的相关字符串存在时才可以访问。

另外，还需要注意容器 `boost::smatch` 的第一个元素存储的引用是指向匹配正则表达式的整个字符串的，匹配第一组的第一个子串由索引 1 访问。

Boost.Regex 提供的第三个函数是 `boost::regex_replace()` 。

```
#include <boost/regex.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = " Boris Schäling ";
  boost::regex expr("\\s");
  std::string fmt("_");
  std::cout << boost::regex_replace(s, expr, fmt) << std::endl;
}
```

- 下载源代码

除了待搜索的字符串和正则表达式之外，`boost::regex_replace()` 函数还需要一个格式参数，它决定了子串、匹配正则表达式的分组如何被替换。如果正则表达式不包含任何分组，相关子串将被用给定的格式一个个地被替换。这样上面程序输出的结果为 `_Boris_Schäling_` 。

`boost::regex_replace()` 函数总是在整个字符串中搜索正则表达式，所以这个程序实际上将三处空格都替换为下划线。

```
#include <boost/regex.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  boost::regex expr("(\\w+)\\s(\\w+)");
  std::string fmt("\\2 \\1");
  std::cout << boost::regex_replace(s, expr, fmt) << std::endl;
}
```

- 下载源代码

格式参数可以访问由正则表达式分组的子串，这个例子正是使用了这项技术，交换了姓、名的位置，于是结果显示为 `Schäling Boris` 。

需要注意的是，对于正则表达式和格式有不同的标准。这三个函数都可以接受一个额外的参数，用于选择具体的标准。也可以指定是否以某一具体格式解释特殊字符或者替代匹配正则表达式的整个字符串。

```
#include <boost/regex.hpp>
#include <locale>
#include <iostream>

int main()
{
  std::locale::global(std::locale("German"));
  std::string s = "Boris Schäling";
  boost::regex expr("(\\w+)\\s(\\w+)");
  std::string fmt("\\2 \\1");
  std::cout << boost::regex_replace(s, expr, fmt, boost::regex_cons
}
```

- 下载源代码

此程序将 `boost::regex_constants::format_literal` 标志作为第四参数传递给函数 `boost::regex_replace()` ，从而抑制了格式参数中对特殊字符的处理。因为整个字符串匹配正则表达式，所以本例中经格式参数替换的到达的输出结果为 `\2 \1` 。

正如上一节末指出的那样，正则表达式可以和 Boost.StringAlgorithms 库结合使用。它通过 Boost.Regex 库提供函数如 `boost::algorithm::find_regex()` 、 `boost::algorithm::replace_regex()` 、 `boost::algorithm::erase_regex()` 以及 `boost::algorithm::split_regex()` 等等。由于 Boost.Regex 库很有可能成为即将到来的下一版 C++ 标准的一部分，脱离 Boost.StringAlgorithms 库，熟练地使用正则表达式是个明智的选择。

## 5.5. 词汇分割器库 Boost.Tokenizer

Boost.Tokenizer 库可以在指定某个字符为分隔符后，遍历字符串的部分表达式。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char> > tokenizer;
  std::string s = "Boost C++ libraries";
  tokenizer tok(s);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::cout << *it << std::endl;
}
```

- 下载源代码

Boost.Tokenizer 库在 `boost/tokenizer.hpp` 文件中定义了模板类 `boost::tokenizer` ，其模板参数为支持相关表达式的类。 上面的例子中就使用了 `boost::char_separator` 类作为模板参数，它将空格和标点符号视为分隔符。

词汇分割器必须由类型为 `std::string` 的字符串初始化。通过使用 `begin()` 和 `end()` 方法，词汇分割器可以像容器一样访问。通过使用迭代器，可以得到前述字符串的部分表达式。模板参数的类型决定了如何达到部分表达式。

因为 `boost::char_separator` 类默认将空格和标点符号视为分隔符，所以本例显示的结果为 `Boost` 、 `C` 、 `+` 、 `+` 和 `libraries` 。 为了识别这些分隔符， `boost::char_separator` 函数调用了 `std::isspace()` 函数和 `std::ispunct` 函数。 () Boost.Tokenizer 库会区分要隐藏的分隔符和要显示的分隔符。 在默认的情况下，空格会隐藏而标点符号会显示出来，所以这个例子里显示了两个加号。

如果不需要将标点符号作为分隔符，可以在传递给词汇分割器之前相应地初始化 `boost::char_separator` 对象。 以下例子正式这样做的。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char> > tokenizer;
  std::string s = "Boost C++ libraries";
  boost::char_separator<char> sep(" ");
  tokenizer tok(s, sep);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::cout << *it << std::endl;
}
```

- 下载源代码

类 `boost::char_separator` 的构造函数可以接受三个参数， 只有第一个是必须的，它描述了需要隐藏的分隔符。 在本例中， 空格仍然被视为分隔符。

第二个参数指定了需要显示的分隔符。 在不提供此参数的情况下，将不显示任何分隔符。 执行程序，会显示 `Boost` 、 `C++` 和 `libraries` 。

如果将加号作为第二个参数，此例的结果将和上一个例子相同。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char> > tokenizer
  std::string s = "Boost C++ libraries";
  boost::char_separator<char> sep(" ", "+");
  tokenizer tok(s, sep);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it
    std::cout << *it << std::endl;
}
```

- 下载源代码

第三个参数决定了是否显示空的部分表达式。 如果连续找到两个分隔符，他们之间的部分表达式将为空。在默认情况下，这些空表达式是不会显示的。第三个参数可以改变默认的行为。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char> > tokenizer
  std::string s = "Boost C++ libraries";
  boost::char_separator<char> sep(" ", "+", boost::keep_empty_toker
  tokenizer tok(s, sep);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it
    std::cout << *it << std::endl;
}
```

- 下载源代码

执行以上程序，会显示另外两个的空表达式。 其中第一个是在两个加号中间的而第二个是加号和之后的空格之间的。

词汇分割器也可用于不同的字符串类型。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<wchar_t>, std::wst
  std::wstring s = L"Boost C++ libraries";
  boost::char_separator<wchar_t> sep(L" ");
  tokenizer tok(s, sep);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::wcout << *it << std::endl;
}
```

- 下载源代码

这个例子遍历了一个类型为 `std::wstring` 的字符串。 为了使用这个类型的字符串，必须使用另外的模板参数初始化词汇分割器，对 `boost::char_separator` 类也是如此，他们都需要参数 `wchar_t` 初始化。

除了 `boost::char_separator` 类之外， Boost.Tokenizer 还提供了另外两个类以识别部分表达式。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::escaped_list_separator<char> > to
  std::string s = "Boost,\"C++ libraries\"";
  tokenizer tok(s);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::cout << *it << std::endl;
}
```

- 下载源代码

`boost::escaped_list_separator` 类用于读取由逗号分隔的多个值，这个格式的文件通常称为 CSV （comma separated values，逗号分隔文件），它甚至还可以处理双引号以及转义序列。所以本例的输出为 `Boost` 和 `C++ libraries` 。

另一个是 `boost::offset_separator` 类，必须用实例说明。 这个类的对象必须作为第二个参数传递给 `boost::tokenizer` 类的构造函数。

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::offset_separator> tokenizer;
  std::string s = "Boost C++ libraries";
  int offsets[] = { 5, 5, 9 };
  boost::offset_separator sep(offsets, offsets + 3);
  tokenizer tok(s, sep);
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::cout << *it << std::endl;
}
```

- 下载源代码

`boost::offset_separator` 指定了部分表达式应当在字符串中的哪个位置结束。 以上程序制定第一个部分表达式在 5 个字符后结束，第二个字符串在另 5 个字符后结束，第三个也就是最后一个字符串应当在之后的 9 个字符后结束。 输出的结果为 `Boost` 、 `C++` 和 `libraries` 。

## 5.6. 格式化输出库 Boost.Format

Boost.Format 库可以作为定义在文件 `cstdio` 中的函数 `std::printf()` 的替代。 `std::printf()` 函数最初出现在 C 标准中，提供格式化数据输出功能， 但是它既不是类型安全的有不能扩展。 因此在 C++ 应用中， Boost.Format 库通常是数据格式化输出的上佳之选。

Boost.Format 库在文件 `boost/format.hpp` 中定义了类 `boost::format` 。 与函数 `std::printf` 相似的是，传递给() `boost::format` 的构造函数的参数也是一个字符串，它由控制格式的特殊字符组成。 实际数据通过操作符 % 相连，在输出中替代特殊字符，如下例所示。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%1%.%2%.%3%") % 16 % 9 % 2008 << std
}
```

- 下载源代码

Boost.Format 类使用置于两个百分号之间的数字作为占位符，占位符稍后通过 % 操作符与实际数据连接。 以上程序使用数字16、9 和 2009 组成一个日期字符串，以 `16.9.2008` 的格式输出。 如果要月份出现在日期之前，即美式表示，只需交换占位符即可。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%2%/%1%/%3%") % 16 % 9 % 2008 << std
}
```

- 下载源代码

现在程序显示的结果变成 `9/16/2008` 。

如果要使用C++ 操作器格式化数据，Boost.Format 库提供了函数 `boost::io::group()` 。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%1% %2% %1%") % boost::io::group(std
}
```

- 下载源代码

本例的结果显示为 `+99 100 +99` 。 因为操作器 `std::showpos()` 通过 `boost::io::group()` 与数字 99 连接，所以只要显示 99 ， 在它前面就会自动加上加号。

如果需要加号仅在 99 第一次输出时显示， 则需要改造格式化占位符。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%|1$+| %2% %1%") % 99 % 100 << std::e
}
```

为了将输出格式改为 `+99 100 99` ，不但需要将数据的引用符号由 1$ 变为 1% ，还需要在其两侧各添加一个附加的管道符号，即将占位符 %1% 替换为 %|1$+|。

请注意，虽然一般对数据的引用不是必须的，但是所有占位符一定要同时设置为指定货非指定。 以下例子在执行时会出现错误，因为它给第二个和第三个占位符设置了引用但是却忽略了第一个。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::format("%|+| %2% %1%") % 99 % 100 << std::e
  }
  catch (boost::io::format_error &ex)
  {
    std::cout << ex.what() << std::endl;
  }
}
```

- 下载源代码

此程序抛出了类型为 `boost::io::format_error` 的异常。 严格地说，Boost.Format 库抛出的异常为 `boost::io::bad_format_string` 。 但是由于所有的异常类都继承自 `boost::io::format_error` 类，捕捉此类型的异常会轻松一些。

以下例子演示了不引用数据的方法。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%|+| %|| %||") % 99 % 100 % 99 << std
}
```

- 下载源代码

第二、第三个占位符的管道符号可以被安全地省略，因为在这种情况下，他们并不指定格式。这样的语法看起来很像 `std::printf ()` 的那种。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%+d %d %d") % 99 % 100 % 99 << std::e
}
```

- 下载源代码

虽然这看起来就像 `std::printf()` ，但是 Boost.Format 库有类型安全的优点。格式化字符串中字母 'd' 的使用并不表示输出数字，而是表示 `boost::format` 类所使用的内部流对象上的 `std::dec()` 操作器，它可以使用某些对 `std::printf()` 函数无意义的格式字符串，如果使用 `std::printf()` 会导致程序在运行时崩溃。

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format("%+s %s %s") % 99 % 100 % 99 << std::e
}
```

- 下载源代码

尽管在 `std::printf()` 函数中，字母 's' 只用于表示类型为 `const char*` 的字符串，然而以上程序却能正常运行。 因为在 Boost.Format 库中，这并不代表强制为字符串，它会结合适当的操作器，调整内部流的操作模式。 所以即使在这种情况下， 在内部流中加入数字也是没问题的。

## 5.7. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 编写程序，从以下 XML 流中提取并显示数据，包括姓名、生日以及账户余额。

   ```
   &lt;person&gt;&lt;name&gt;Karl-Heinz Huber&lt;/name&gt;&lt;dob&g
   ```

   。

   姓、名要分开显示，生日使用 “日.月.年” 的格式，账户余额忽略小数位。 使用其他 XML 流测试你的程序，如包含多余空白、其他名字、账户余额为负数等等的 XML 流。

2. 编写程序，使得格式与显示的数据记录如下：输入 **Munich Hamburg 92.12 8:25 9:45** ， 这条记录表示从 Munich 到 Hamburg 的航班票价为 92.12 欧元，上午 8:25 起飞 9:45 到达目的地。要得到以下输出 Munich -> Hamburg 92.12 EUR (08:25-09:45) 。

   具体地说，城市名称长度为10并且左对齐而票价长度为7并且右对齐，货币在价格后显示。 起飞与降落时间一起显示在圆括号中，以连字符分隔，不留空格。对早于10点（上午或下午）的时间，必须在前面补0。 用不同的数据记录测试你的程序，例如使用长度大于10的城市名。

# 第 6 章 多线程

## 目录






## 6.1. 概述

线程就是，在同一程序同一时间内允许执行不同函数的离散处理队列。 这使得一个长时间去进行某种特殊运算的函数在执行时不阻碍其他的函数变得十分重要。 线程实际上允许同时执行两种函数，而这两个函数不必相互等待。

一旦一个应用程序启动，它仅包含一个默认线程。 此线程执行 `main()` 函数。 在 `main()` 中被调用的函数则按这个线程的上下文顺序地执行。 这样的程序称为单线程程序。

反之，那些创建新的线程的程序就是多线程程序。 他们不仅可以在同一时间执行多个函数，而且这在如今多核盛行的时代显得尤为重要。 既然多核允许同时执行多个函数，这就使得对开发人员相应地使用这种处理能力提出了要求。 然而线程一直被用来当并发地执行多个函数，开发人员现在不得不仔细地构建应用来支持这种并发。 多线程编程知识也因此在多核系统时代变得越来越重要。

本章将介绍C++ Boost库 Boost.Thread，它可以开发独立于平台的多线程应用程序。

## 6.2. 线程管理

在这个库最重要的一个类就是 `boost::thread` ，它是在 `boost/thread.hpp` 里定义的，用来创建一个新线程。下面的示例来说明如何运用它。

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << std::endl;
  }
}

int main()
{
  boost::thread t(thread);
  t.join();
}
```

- 下载源代码

新建线程里执行的那个函数的名称被传递到 `boost::thread` 的构造函数。 一旦上述示例中的变量 `t` 被创建，该 `thread()` 函数就在其所在线程中被立即执行。 同时在 `main()` 里也并发地执行该 `thread()` 。

为了防止程序终止，就需要对新建线程调用 `join()` 方法。 `join()` 方法是一个阻塞调用：它可以暂停当前线程，直到调用 `join()` 的线程运行结束。 这就使得 `main()` 函数一直会等待到 `thread()` 运行结束。

正如在上面的例子中看到，一个特定的线程可以通过诸如 `t` 的变量访问，通过这个变量等待着它的使用 `join()` 方法终止。 但是，即使 `t` 越界或者析构了，该线程也将继续执行。 一个线程总是在一开始就绑定到一个类型为 `boost::thread` 的变量，但是一旦创建，就不在取决于它。 甚至还存在着一个叫 `detach()` 的方法，允许类型为 `boost::thread` 的变量从它对应的线程里分离。 当然了，像 `join()` 的方法之后也就不能被调用，因为这个变量不再是一个有效的线程。

任何一个函数内可以做的事情也可以在一个线程内完成。 归根结底，一个线程只不过是一个函数，除了它是同时执行的。 在上述例子中，使用一个循环把5个数字写入标准输出流。 为了减缓输出，每一个循环中调用 `wait()` 函数让执行延迟了一秒。 `wait()` 可以调用一个名为 `sleep()` 的函数，这个函数也来自于 Boost.Thread，位于 `boost::this_thread` 名空间内。

`sleep()` 要么在预计的一段时间或一个特定的时间点后时才让线程继续执行。 通过传递一个类型为 `boost::posix_time::seconds` 的对象，在这个例子里我们指定了一段时间。 `boost::posix_time::seconds` 来自于 Boost.DateTime 库，它被 Boost.Thread 用来管理和处理时间的数据。

虽然前面的例子说明了如何等待一个不同的线程，但下面的例子演示了如何通过所谓的中断点让一个线程中断。

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

void thread()
{
  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << std::endl;
    }
  }
  catch (boost::thread_interrupted&)
  {
  }
}

int main()
{
  boost::thread t(thread);
  wait(3);
  t.interrupt();
  t.join();
}
```

- 下载源代码

在一个线程对象上调用 `interrupt()` 会中断相应的线程。 在这方面，中断意味着一个类型为 `boost::thread_interrupted` 的异常，它会在这个线程中抛出。 然后这只有在线程达到中断点时才会发生。

如果给定的线程不包含任何中断点，简单调用 `interrupt()` 就不会起作用。 每当一个线程中断点，它就会检查 `interrupt()` 是否被调用过。 只有被调用过了， `boost::thread_interrupted` 异常才会相应地抛出。

Boost.Thread定义了一系列的中断点，例如 `sleep()` 函数。 由于 `sleep()` 在这个例子里被调用了五次，该线程就检查了五次它是否应该被中断。 然而 `sleep()` 之间的调用，却不能使线程中断。

一旦该程序被执行，它只会打印三个数字到标准输出流。 这是由于在main里3秒后调用 `interrupt()` 方法。 因此，相应的线程被中断，并抛出一个 `boost::thread_interrupted` 异常。 这个异常在线程内也被正确地捕获， `catch` 处理虽然是空的。 由于 `thread()` 函数在处理程序后返回，线程也被终止。 这反过来也将终止整个程序，因为 `main()` 等待该线程使用join（）终止该线程。

Boost.Thread定义包括上述 `sleep()` 函数十个中断。 有了这些中断点，线程可以很容易及时中断。 然而，他们并不总是最佳的选择，因为中断点必须事前读入以检查 `boost::thread_interrupted` 异常。

为了提供一个对 Boost.Thread 里提供的多种函数的整体概述，下面的例子将会再介绍两个。

```
#include <boost/thread.hpp>
#include <iostream>

int main()
{
  std::cout << boost::this_thread::get_id() << std::endl;
  std::cout << boost::thread::hardware_concurrency() << std::endl;
}
```

- 下载源代码

使用 `boost::this_thread` 命名空间，能提供独立的函数应用于当前线程，比如前面出现的 `sleep()` 。 另一个是 `get_id()` ：它会返回一个当前线程的ID号。 它也是由 `boost::thread` 提供的。

`boost::thread` 类提供了一个静态方法 `hardware_concurrency()` ，它能够返回基于CPU数目或者CPU内核数目的刻在同时在物理机器上运行的线程数。 在常用的双核机器上调用这个方法，返回值为2。 这样的话就可以确定在一个多核程序可以同时运行的理论最大线程数。

## 6.3. 同步

虽然多线程的使用可以提高应用程序的性能，但也增加了复杂性。 如果使用线程在同一时间执行几个函数，访问共享资源时必须相应地同步。 一旦应用达到了一定规模，这涉及相当一些工作。 本段介绍了Boost.Thread提供同步线程的类。

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
    std::cout << "Thread " << boost::this_thread::get_id() << ": "
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

- 下载源代码

多线程程序使用所谓的互斥对象来同步。 Boost.Thread提供多个的互斥类， `boost::mutex` 是最简单的一个。 互斥的基本原则是当一个特定的线程拥有资源的时候防止其他线程夺取其所有权。 一旦释放，其他的线程可以取得所有权。这将导致线程等待至另一个线程完成处理一些操作，从而相应地释放互斥对象的所有权。

上面的示例使用一个类型为 `boost::mutex` 的 `mutex` 全局互斥对象。 `thread()` 函数获取此对象的所有权才在 `for` 循环内使用 `lock()` 方法写入到标准输出流的。 一旦信息被写入，使用 `unlock()` 方法释放所有权。

`main()` 创建两个线程，同时执行 `thread ()` 函数。 利用 `for` 循环，每个线程数到5，用一个迭代器写一条消息到标准输出流。 不幸的是，标准输出流是一个全局性的被所有线程共享的对象。 该标准不提供任何保证 `std::cout` 可以安全地从多个线程访问。 因此，访问标准输出流必须同步：在任何时候，只有一个线程可以访问 `std::cout` 。

由于两个线程试图在写入标准输出流前获得互斥体，实际上只能保证一次只有一个线程访问 `std::cout` 。不管哪个线程成功调用 `lock()` 方法，其他所有线程必须等待，直到 `unlock()` 被调用。

获取和释放互斥体是一个典型的模式，是由Boost.Thread通过不同的数据类型支持。例如，不直接地调用 `lock()` 和 `unlock()` ，使用 `boost::lock_guard` 类也是可以的。

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::lock_guard<boost::mutex> lock(mutex);
    std::cout << "Thread " << boost::this_thread::get_id() << ": "
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

- 下载源代码

`boost::lock_guard` 在其内部构造和析构函数分别自动调用 `lock()` 和 `unlock()` 。访问共享资源是需要同步的，因为它显示地被两个方法调用。 `boost::lock_guard` 类是另一个出现在 第 2 章 智能指针 的RAII用语。

除了 `boost::mutex` 和 `boost::lock_guard` 之外，Boost.Thread也提供其他的类支持各种同步。其中一个重要的就是 `boost::unique_lock` ，相比较 `boost::lock_guard` 而言，它提供许多有用的方法。

```
#include <boost/thread.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::timed_mutex mutex;

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::unique_lock<boost::timed_mutex> lock(mutex, boost::try_t
    if (!lock.owns_lock())
      lock.timed_lock(boost::get_system_time() + boost::posix_time
    std::cout << "Thread " << boost::this_thread::get_id() << ": "
    boost::timed_mutex *m = lock.release();
    m->unlock();
  }
}

int main()
{
  boost::thread t1(thread);
  boost::thread t2(thread);
  t1.join();
  t2.join();
}
```

- 下载源代码

上面的例子用不同的方法来演示 `boost::unique_lock` 的功能。 当然了，这些功能的用法对给定的情景不一定适用； `boost::lock_guard` 在上个例子的用法还是挺合理的。 这个例子就是为了演示 `boost::unique_lock` 提供的功能。

`boost::unique_lock` 通过多个构造函数来提供不同的方式获得互斥体。 这个期望获得互斥体的函数简单地调用了 `lock()` 方法，一直等到获得这个互斥体。 所以它的行为跟 `boost::lock_guard` 的那个是一样的。

如果第二个参数传入一个 `boost::try_to_lock` 类型的值，对应的构造函数就会调用 `try_lock()` 方法。 这个方法返回 `bool` 型的值：如果能够获得互斥体则返回 `true` ，否则返回 `false` 。 相比 `lock()` 函数， `try_lock()` 会立即返回，而且在获得互斥体之前不会被阻塞。

上面的程序向 `boost::unique_lock` 的构造函数的第二个参数传入 `boost::try_to_lock` 。 然后通过 `owns_lock()` 可以检查是否可获得互斥体。 如果不能， `owns_lock()` 返回 `false` 。 这也用到 `boost::unique_lock` 提供的另外一个函数： `timed_lock()` 等待一定的时间以获得互斥体。 给定的程序等待长达1秒，应较足够的时间来获取更多的互斥。

其实这个例子显示了三个方法获取一个互斥体： `lock()` 会一直等待，直到获得一个互斥体。 `try_lock()` 则不会等待，但如果它只会在互斥体可用的时候才能获得，否则返回 `false` 。 最后， `timed_lock()` 试图获得在一定的时间内获取互斥体。 和 `try_lock()` 一样，返回 `bool` 类型的值意味着成功是否。

虽然 `boost::mutex` 提供了 `lock()` 和 `try_lock()` 两个方法，但是 `boost::timed_mutex` 只支持 `timed_lock()` ，这就是上面示例那么使用的原因。 如果不用 `timed_lock()` 的话，也可以像以前的例子那样用 `boost::mutex` 。

就像 `boost::lock_guard` 一样， `boost::unique_lock` 的析构函数也会相应地释放互斥量。此外，可以手动地用 `unlock()` 释放互斥量。也可以像上面的例子那样，通过调用 `release()` 解除 `boost::unique_lock` 和互斥量之间的关联。然而在这种情况下，必须显式地调用 `unlock()` 方法来释放互斥量，因为 `boost::unique_lock` 的析构函数不再做这件事情。

`boost::unique_lock` 这个所谓的独占锁意味着一个互斥量同时只能被一个线程获取。 其他线程必须等待，直到互斥体再次被释放。 除了独占锁，还有非独占锁。 Boost.Thread里有个 `boost::shared_lock` 的类提供了非独占锁。 正如下面的例子，这个类必须和 `boost::shared_mutex` 型的互斥量一起使用。

```
#include <boost/thread.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

void wait(int seconds)
{
  boost::this_thread::sleep(boost::posix_time::seconds(seconds));
}

boost::shared_mutex mutex;
std::vector<int> random_numbers;

void fill()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::shared_mutex> lock(mutex);
    random_numbers.push_back(std::rand());
    lock.unlock();
    wait(1);
```

```
    }
  }

  void print()
  {
    for (int i = 0; i < 3; ++i)
    {
      wait(1);
      boost::shared_lock<boost::shared_mutex> lock(mutex);
      std::cout << random_numbers.back() << std::endl;
    }
  }

  int sum = 0;

  void count()
  {
    for (int i = 0; i < 3; ++i)
    {
      wait(1);
      boost::shared_lock<boost::shared_mutex> lock(mutex);
      sum += random_numbers.back();
    }
  }

  int main()
  {
    boost::thread t1(fill);
    boost::thread t2(print);
    boost::thread t3(count);
    t1.join();
    t2.join();
    t3.join();
    std::cout << "Sum: " << sum << std::endl;
  }
```

- 下载源代码

`boost::shared_lock` 类型的非独占锁可以在线程只对某个资源读访问的情况下使用。 一个线程修改的资源需要写访问，因此需要一个独占锁。 这样做也很明显：只需要读访问的线程不需要知道同一时间其他线程是否访问。 因此非独占锁可以共享一个互斥体。

在给定的例子， `print()` 和 `count()` 都可以只读访问 `random_numbers` 。 虽然 `print()` 函数把 `random_numbers` 里的最后一个数写到标准输出， `count()` 函数把它统计到 `sum` 变量。 由于没有函数修改 `random_numbers` ，所有的都可以在同一时间用 `boost::shared_lock` 类型的非独占锁访问它。

在 `fill()` 函数里，需要用一个 `boost::unique_lock` 类型的非独占锁，因为它插入了一个新的随机数到 `random_numbers` 。在 `unlock()` 显式地调用 `unlock()` 来释放互斥量之后， `fill()` 等待了一秒。相比于之前的那个样子，在 `for` 循环的尾部调用 `wait()` 以保证容器里至少存在一个随机数，可以被 `print()` 或者 `count()` 访问。对应地，这两个函数在 `for` 循环的开始调用了 `wait()` 。

考虑到在不同的地方每个单独地调用 `wait()` ，一个潜在的问题变得很明显:函数调用的顺序直接受CPU执行每个独立进程的顺序决定。利用所谓的条件变量，可以同步哪些独立的线程，使数组的每个元素都被不同的线程立即添加到 `random_numbers` 。

```
#include <boost/thread.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

boost::mutex mutex;
boost::condition_variable_any cond;
std::vector<int> random_numbers;

void fill()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::mutex> lock(mutex);
    random_numbers.push_back(std::rand());
    cond.notify_all();
    cond.wait(mutex);
  }
}

void print()
{
  std::size_t next_size = 1;
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::mutex> lock(mutex);
    while (random_numbers.size() != next_size)
      cond.wait(mutex);
    std::cout << random_numbers.back() << std::endl;
    ++next_size;
    cond.notify_all();
  }
}

int main()
{
  boost::thread t1(fill);
  boost::thread t2(print);
  t1.join();
  t2.join();
}
```

- 下载源代码

这个例子的程序删除了 `wait()` 和 `count()` 。线程不用在每个循环迭代中等待一秒，而是尽可能快地执行。此外，没有计算总额；数字完全写入标准输出流。

为确保正确地处理随机数，需要一个允许检查多个线程之间特定条件的条件变量来同步不每个独立的线程。

正如上面所说， `fill()` 函数用在每个迭代产生一个随机数，然后放在 `random_numbers` 容器中。 为了防止其他线程同时访问这个容器，就要相应得使用一个排它锁。 不是等待一秒，实际上这个例子却用了一个条件变量。 调用 `notify_all()` 会唤醒每个哪些正在分别通过调用 `wait()` 等待此通知的线程。

通过查看 `print()` 函数里的 `for` 循环，可以看到相同的条件变量被 `wait()` 函数调用了。 如果这个线程被 `notify_all()` 唤醒，它就会试图这个互斥量，但只有在 `fill()` 函数完全释放之后才能成功。

这里的窍门就是调用 `wait()` 会释放相应的被参数传入的互斥量。 在调用 `notify_all()` 后， `fill()` 函数会通过 `wait()` 相应地释放线程。 然后它会阻止和等待其他的线程调用 `notify_all()` ，一旦随机数已写入标准输出流，这就会在 `print()` 里发生。

注意到在 `print()` 函数里调用 `wait()` 事实上发生在一个单独 `while` 循环里。 这样做的目的是为了处理在 `print()` 函数里第一次调用 `wait()` 函数之前随机数已经放到容器里。 通过比较 `random_numbers` 里元素的数目与预期值，发现这成功地处理了把随机数写入到标准输出流。

## 6.4. 线程本地存储

线程本地存储（TLS）是一个只能由一个线程访问的专门的存储区域。 TLS的变量可以被看作是一个只对某个特定线程而非整个程序可见的全局变量。 下面的例子显示了这些变量的好处。

```
#include <boost/thread.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>

void init_number_generator()
{
  static bool done = false;
  if (!done)
  {
    done = true;
    std::srand(static_cast<unsigned int>(std::time(0)));
  }
}

boost::mutex mutex;

void random_number_generator()
{
  init_number_generator();
  int i = std::rand();
  boost::lock_guard<boost::mutex> lock(mutex);
  std::cout << i << std::endl;
}

int main()
{
  boost::thread t[3];

  for (int i = 0; i < 3; ++i)
    t[i] = boost::thread(random_number_generator);

  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

- 下载源代码

该示例创建三个线程，每个线程写一个随机数到标准输出流。 `random_number_generator()` 函数将会利用在C++标准里定义的 `std::rand()` 函数创建一个随机数。 但是用于 `std::rand()` 的随机数产生器必须先用 `std::srand()` 正确地初始化。 如果没做，程序始终打印同一个随机数。

随机数产生器，通过 `std::time()` 返回当前时间， 在 `init_number_generator()` 函数里完成初始化。 由于这个值每次都不同，可以保证产生器总是用不同的值初始化，从而产生不同的随机数。 因为产生器只要初始化一次， `init_number_generator()` 用了一个静态变量 `done` 作为条件量。

如果程序运行了多次，写入的三分之二的随机数显然就会相同。 事实上这个程序有个缺陷： `std::rand()` 所用的产生器必须被各个线程初始化。 因此 `init_number_generator()` 的实现实际上是不对的，因为它只调用了一次 `std::srand()` 。使用TLS，这一缺陷可以得到纠正。

```
#include <boost/thread.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>

void init_number_generator()
{
  static boost::thread_specific_ptr<bool> tls;
  if (!tls.get())
    tls.reset(new bool(false));
  if (!*tls)
  {
    *tls = true;
    std::srand(static_cast<unsigned int>(std::time(0)));
  }
}

boost::mutex mutex;

void random_number_generator()
{
  init_number_generator();
  int i = std::rand();
  boost::lock_guard<boost::mutex> lock(mutex);
  std::cout << i << std::endl;
}

int main()
{
  boost::thread t[3];

  for (int i = 0; i < 3; ++i)
    t[i] = boost::thread(random_number_generator);

  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

- 下载源代码

用一个TLS变量 `tls` 代替静态变量 `done` ，是基于用 `bool` 类型实例化的 `boost::thread_specific_ptr` 。 原则上， `tls` 工作起来就像 `done` ：它可以作为一个条件指明随机数发生器是否被初始化。 但是关键的区别，就是 `tls` 存储的值只对相应的线程可见和可用。

一旦一个 boost::thread_specific_ptr 型的变量被创建，它可以相应地设置。 不过，它期望得到一个 bool 型变量的地址，而非它本身。使用 reset() 方法，可以把它的地址保存到 tls 里面。 在给出的例子中，会动态地分配一个 bool 型的变量，由 new 返回它的地址，并保存到 tls 里。 为了避免每次调用 init_number_generator() 都设置 tls ，它会通过 get() 函数检查是否已经保存了一个地址。

由于 boost::thread_specific_ptr 保存了一个地址，它的行为就像一个普通的指针。 因此， operator*() 和 operator->() 都被被重载以方便使用。 这个例子用 *tls 检查这个条件当前是 true 还是 false 。 再根据当前的条件，随机数生成器决定是否初始化。

正如所见， boost::thread_specific_ptr 允许为当前进程保存一个对象的地址，然后只允许当前进程获得这个地址。 然而，当一个线程已经成功保存这个地址，其他的线程就会可能就失败。

如果程序正在执行时，它可能会令人感到奇怪：尽管有了TLS的变量，生成的随机数仍然相等。 这是因为，三个线程在同一时间被创建，从而造成随机数生成器在同一时间初始化。 如果该程序执行了几次，随机数就会改变，这就表明生成器初始化正确了。

## 6.5. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 重构下面的程序用两个线程来计算总和。由于现在许多处理器有两个内核，应利用线程减少执行时间。

```
#include &lt;boost/date_time/posix_time/posix_time.hpp&gt;
#include &lt;boost/cstdint.hpp&gt;
#include &lt;iostream&gt;

int main()
{
  boost::posix_time::ptime start = boost::posix_time::microsec_

  boost::uint64_t sum = 0;
  for (int i = 0; i &lt; 1000000000; ++i)
    sum += i;

  boost::posix_time::ptime end = boost::posix_time::microsec_cl
  std::cout &lt;&lt; end - start &lt;&lt; std::endl;

  std::cout &lt;&lt; sum &lt;&lt; std::endl;
}
```

   - 下载源代码

2. 通过利用处理器尽可能同时执行多的线程，把例1一般化。 例如，如果处理器有四个内核，就应该利用四个线程。

3. 修改下面的程序，在 `main()` 中自己的线程中执行 `thread()` 。 程序应该能够计算总和，然后把结果输入到标准输出两次。 但可以更改 `calculate()` ， `print()` 和 `thread()` 的实现，每个函数的接口仍需保持一致。 也就是说每个函数应该仍然没有任何参数，也不需要返回一个值。

   ```
   #include &lt;iostream&gt;

   int sum = 0;

   void calculate()
   {
     for (int i = 0; i &lt; 1000; ++i)
       sum += i;
   }

   void print()
   {
     std::cout &lt;&lt; sum &lt;&lt; std::endl;
   }

   void thread()
   {
     calculate();
     print();
   }

   int main()
   {
     thread();
   }
   ```

   - 下载源代码

# 第 7 章 异步输入输出

目录






## 7.1. 概述

本章介绍了 Boost C++ 库 Asio，它是异步输入输出的核心。 名字本身就说明了一切：Asio 意即异步输入/输出。 该库可以让 C++ 异步地处理数据，且平台独立。 异步数据处理就是指，任务触发后不需要等待它们完成。 相反，Boost.Asio 会在任务完成时触发一个应用。 异步任务的主要优点在于，在等待任务完成时不需要阻塞应用程序，可以去执行其它任务。

异步任务的典型例子是网络应用。 如果数据被发送出去了，比如发送至 Internet，通常需要知道数据是否发送成功。 如果没有一个象 Boost.Asio 这样的库，就必须对函数的返回值进行求值。 但是，这样就要求待至所有数据发送完毕，并得到一个确认或是错误代码。 而使用 Boost.Asio，这个过程被分为两个单独的步骤：第一步是作为一个异步任务开始数据传输。 一旦传输完成，不论成功或是错误，应用程序都会在第二步中得到关于相应的结果通知。 主要的区别在于，应用程序无需阻塞至传输完成，而可以在这段时间里执行其它操作。

## 7.2. I/O 服务与 I/O 对象

使用 Boost.Asio 进行异步数据处理的应用程序基于两个概念：I/O 服务和 I/O 对象。 I/O 服务抽象了操作系统的接口，允许第一时间进行异步数据处理，而 I/O 对象则用于初始化特定的操作。 鉴于 Boost.Asio 只提供了一个名为 `boost::asio::io_service` 的类作为 I/O 服务，它针对所支持的每一个操作系统都分别实现了优化的类，另外库中还包含了针对不同 I/O 对象的几个类。 其中，类 `boost::asio::ip::tcp::socket` 用于通过网络发送和接收数据，而类 `boost::asio::deadline_timer` 则提供了一个计时器，用于测量某个固定时间点到来或是一段指定的时长过去了。 以下第一个例子中就使用了计时器，因为与 Asio 所提供的其它 I/O 对象相比较而言，它不需要任何有关于网络编程的知识。

```
#include <boost/asio.hpp>
#include <iostream>

void handler(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

int main()
{
  boost::asio::io_service io_service;
  boost::asio::deadline_timer timer(io_service, boost::posix_time::
  timer.async_wait(handler);
  io_service.run();
}
```

- 下载源代码

函数 `main()` 首先定义了一个 I/O 服务 `io_service`，用于初始化 I/O 对象 `timer`。 就象 `boost::asio::deadline_timer` 那样，所有 I/O 对象通常都需要一个 I/O 服务作为它们的构造函数的第一个参数。 由于 `timer` 的作用类似于一个闹钟，所以 `boost::asio::deadline_timer` 的构造函数可以传入第二个参数，用于表示在某个时间点或是在某段时长之后闹钟停止。 以上例子指定了五秒的时长，该闹钟在 `timer` 被定义之后立即开始计时。

虽然我们可以调用一个在五秒后返回的函数，但是通过调用方法 `async_wait()` 并传入 `handler()` 函数的名字作为唯一参数，可以让 Asio 启动一个异步操作。 请留意，我们只是传入了 `handler()` 函数的名字，而该函数本身并没有被调用。

`async_wait()` 的好处是，该函数调用会立即返回，而不是等待五秒钟。 一旦闹钟时间到，作为参数所提供的函数就会被相应调用。 因此，应用程序可以在调用了 `async_wait()` 之后执行其它操作，而不是阻塞在这里。

象 `async_wait()` 这样的方法被称为是非阻塞式的。 I/O 对象通常还提供了阻塞式的方法，可以让执行流在特定操作完成之前保持阻塞。 例如，可以调用阻塞式的 `wait()` 方法，取代 `boost::asio::deadline_timer` 的调用。 由于它会阻塞调用，所以它不需要传入一个函数名，而是在指定时间点或指定时长之后返回。

再看看上面的源代码，可以留意到在调用 `async_wait()` 之后，又在 I/O 服务之上调用了一个名为 `run()` 的方法。这是必须的，因为控制权必须被操作系统接管，才能在五秒之后调用 `handler()` 函数。

`async_wait()` 会启动一个异步操作并立即返回，而 `run()` 则是阻塞的。因此调用 `run()` 后程序执行会停止。 具有讽刺意味的是，许多操作系统只是通过阻塞函数来支持异步操作。 以下例子显示了为什么这个限制通常不会成为问题。

```
#include <boost/asio.hpp>
#include <iostream>

void handler1(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

void handler2(const boost::system::error_code &ec)
{
  std::cout << "10 s." << std::endl;
}

int main()
{
  boost::asio::io_service io_service;
  boost::asio::deadline_timer timer1(io_service, boost::posix_time
  timer1.async_wait(handler1);
  boost::asio::deadline_timer timer2(io_service, boost::posix_time
  timer2.async_wait(handler2);
  io_service.run();
}
```

- 下载源代码

上面的程序用了两个 `boost::asio::deadline_timer` 类型的 I/O 对象。 第一个 I/O 对象表示一个五秒后触发的闹钟，而第二个则表示一个十秒后触发的闹钟。 每一段指定时长过去后，都会相应地调用函数 `handler1()` 和 `handler2()` 。

在 `main()` 的最后，再次在唯一的 I/O 服务之上调用了 `run()` 方法。 如前所述，这个函数将阻塞执行，把控制权交给操作系统以接管异步处理。 在操作系统的帮助下， `handler1()` 函数会在五秒后被调用，而 `handler2()` 函数则在十秒后被调用。

乍一看，你可能会觉得有些奇怪，为什么异步处理还要调用阻塞式的 `run()` 方法。 然而，由于应用程序必须防止被中止执行，所以这样做实际上不会有任何问题。 如果 `run()` 不是阻塞的， `main()` 就会结束从而中止该应用程序。 如果应用程序不应被阻塞，那么就应该在一个新的线程内部调用 `run()` ，它自然就会仅仅阻塞那个线程。

一旦特定的 I/O 服务的所有异步操作都完成了，控制权就会返回给 `run()` 方法，然后它就会返回。 以上两个例子中，应用程序都会在闹钟到时间后马上结束。

## 7.3. 可扩展性与多线程

用 Boost.Asio 这样的库来开发应用程序，与一般的 C++ 风格不同。 那些可能需要较长时间才返回的函数不再是以顺序的方式来调用。 不再是调用阻塞式的函数，Boost.Asio 是启动一个异步操作。 而那些需要在操作结束后调用的函数则实现为相应的句柄。 这种方法的缺点是，本来顺序执行的功能变得在物理上分割开来了，从而令相应的代码更难理解。

象 Boost.Asio 这样的库通常是为了令应用程序具有更高的效率。 应用程序不需要等待特定的函数执行完成，而可以在期间执行其它任务，如开始另一个需要较长时间的操作。

可扩展性是指，一个应用程序从新增资源有效地获得好处的能力。 如果那些执行时间较长的操作不应该阻塞其它操作的话，那么建议使用 Boost.Asio. 由于现今的PC机通常都具有多核处理器，所以线程的应用可以进一步提高一个基于 Boost.Asio 的应用程序的可扩展性。

如果在某个 `boost::asio::io_service` 类型的对象之上调用 `run()` 方法，则相关联的句柄也会在同一个线程内被执行。 通过使用多线程，应用程序可以同时调用多个 `run()` 方法。 一旦某个异步操作结束，相应的 I/O 服务就将在这些线程中的某一个之中执行句柄。 如果第二个操作在第一个操作之后很快也结束了，则 I/O 服务可以在另一个线程中执行句柄，而无需等待第一个句柄终止。

```
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <iostream>

void handler1(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

void handler2(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

boost::asio::io_service io_service;

void run()
{
  io_service.run();
}

int main()
{
  boost::asio::deadline_timer timer1(io_service, boost::posix_time
  timer1.async_wait(handler1);
  boost::asio::deadline_timer timer2(io_service, boost::posix_time
  timer2.async_wait(handler2);
  boost::thread thread1(run);
  boost::thread thread2(run);
  thread1.join();
  thread2.join();
}
```

- 下载源代码

上一节中的例子现在变成了一个多线程的应用。 通过使用在 `boost/thread.hpp` 中定义的 `boost::thread` 类，它来自于 Boost C++ 库 Thread，我们在 `main()` 中创建了两个线程。 这两个线程均针对同一个 I/O 服务调用了 `run()` 方法。 这样当异步操作完成时，这个 I/O 服务就可以使用两个线程去执行句柄函数。

这个例子中的两个计时数均被设为在五秒后触发。 由于有两个线程，所以 `handler1()` 和 `handler2()` 可以同时执行。 如果第二个计时器触发时第一个仍在执行，则第二个句柄就会在第二个线程中执行。 如果第一个计时器的句柄已经终止，则 I/O 服务可以自由选择任一线程。

线程可以提高应用程序的性能。 因为线程是在处理器内核上执行的，所以创建比内核数更多的线程是没有意义的。 这样可以确保每个线程在其自己的内核上执行，而没有同一内核上的其它线程与之竞争。

要注意，使用线程并不总是值得的。 以上例子的运行会导致不同信息在标准输出流上混合输出，因为这两个句柄可能会并行运行，访问同一个共享资源：标准输出流 `std::cout` 。 这种访问必须被同步，以保证每一条信息在另一个线程可以向标准输出流写出另一条信息之前被完全写出。 在这种情形下使用线程并不能提供多少好处，如果各个独立句柄不能独立地并行运行。

多次调用同一个 I/O 服务的 `run()` 方法，是为基于 Boost.Asio 的应用程序增加可扩展性的推荐方法。 另外还有一个不同的方法：不要绑定多个线程到单个 I/O 服务，而是创建多个 I/O 服务。 然后每一个 I/O 服务使用一个线程。 如果 I/O 服务的数量与系统的处理器内核数量相匹配，则异步操作都可以在各自的内核上执行。

```
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <iostream>

void handler1(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

void handler2(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

boost::asio::io_service io_service1;
boost::asio::io_service io_service2;

void run1()
{
  io_service1.run();
}

void run2()
{
  io_service2.run();
}

int main()
{
  boost::asio::deadline_timer timer1(io_service1, boost::posix_time
  timer1.async_wait(handler1);
  boost::asio::deadline_timer timer2(io_service2, boost::posix_time
  timer2.async_wait(handler2);
  boost::thread thread1(run1);
  boost::thread thread2(run2);
  thread1.join();
  thread2.join();
}
```

- 下载源代码

前面的那个使用两个计时器的例子被重写为使用两个 I/O 服务。 这个应用程序仍然基于两个线程；但是现在每个线程被绑定至不同的 I/O 服务。 此外，两个 I/O 对象 `timer1` 和 `timer2` 现在也被绑定至不同的 I/O 服务。

这个应用程序的功能与前一个相同。 在一定条件下使用多个 I/O 服务是有好处的，每个 I/O 服务有自己的线程，最好是运行在各自的处理器内核上，这样每一个异步操作连同它们的句柄就可以局部化执行。 如果没有远端的数据或函数需要访问，那么每一个 I/O 服务就象一个小的自主应用。 这里的局部和远端是指象高速缓存、内存页这样的资源。 由于在确定优化策略之前需要对底层硬件、操作系统、编译器以及潜在的瓶颈有专门的了解，所以应该仅在清楚这些好处的情况下使用多个 I/O 服务。

## 7.4. 网络编程

虽然 Boost.Asio 是一个可以异步处理任何种类数据的库，但是它主要被用于网络编程。 这是由于，事实上 Boost.Asio 在加入其它 I/O 对象之前很久就已经支持网络功能了。 网络功能是异步处理的一个很好的例子，因为通过网络进行数据传输可能会需要较长时间，从而不能直接获得确认或错误条件。

Boost.Asio 提供了多个 I/O 对象以开发网络应用。 以下例子使用了 `boost::asio::ip::tcp::socket` 类来建立与中另一台PC的连接，并下载 'Highscore' 主页；就象一个浏览器在指向 www.highscore.de 时所要做的。

```
#include <boost/asio.hpp>
#include <boost/array.hpp>
#include <iostream>
#include <string>

boost::asio::io_service io_service;
boost::asio::ip::tcp::resolver resolver(io_service);
boost::asio::ip::tcp::socket sock(io_service);
boost::array<char, 4096> buffer;

void read_handler(const boost::system::error_code &ec, std::size_t
{
  if (!ec)
  {
    std::cout << std::string(buffer.data(), bytes_transferred) << s
    sock.async_read_some(boost::asio::buffer(buffer), read_handler)
  }
}

void connect_handler(const boost::system::error_code &ec)
{
  if (!ec)
  {
    boost::asio::write(sock, boost::asio::buffer("GET / HTTP 1.1\r\
    sock.async_read_some(boost::asio::buffer(buffer), read_handler)
  }
}

void resolve_handler(const boost::system::error_code &ec, boost::as
{
  if (!ec)
  {
    sock.async_connect(*it, connect_handler);
  }
}

int main()
{
  boost::asio::ip::tcp::resolver::query query("www.highscore.de", '
  resolver.async_resolve(query, resolve_handler);
  io_service.run();
}
```

- 下载源代码

这个程序最明显的部分是三个句柄的使用： `connect_handler()` 和 `read_handler()` 函数会分别在连接被建立后以及接收到数据后被调用。 那么为什么需要 `resolve_handler()` 函数呢？

互联网使用了所谓的IP地址来标识每台PC。 IP地址实际上只是一长串数字，难以记住。 而记住象 www.highscore.de 这样的名字就容易得多。 为了在互联网上使用类似的名字，需要通过一个叫作域名解析的过程将它们翻译成相应的IP地址。 这个过程由所谓的域名解析器来完成，对应的 I/O 对象是： `boost::asio::ip::tcp::resolver` 。

域名解析也是一个需要连接到互联网的过程。 有些专门的PC，被称为DNS服务器，其作用就象是电话本，它知晓哪个IP地址被赋给了哪台PC。 由于这个过程本身的透明的，只要明白其背后的概念以及为何需要 `boost::asio::ip::tcp::resolver` I/O 对象就可以了。 由于域名解析不是发生在本地的，所以它也被实现为一个异步操作。 一旦域名解析成功或被某个错误中断， `resolve_handler()` 函数就会被调用。

因为接收数据需要一个成功的连接，进而需要一次成功的域名解析，所以这三个不同的异步操作要以三个不同的句柄来启动。 `resolve_handler()` 访问 I/O 对象 `sock` ，用由迭代器 `it` 所提供的解析后地址创建一个连接。 而 `sock` 也在 `connect_handler()` 的内部被使用，发送 HTTP 请求并启动数据的接收。 因为所有这些操作都是异步的，各个句柄的名字被作为参数传递。 取决于各个句柄，需要相应的其它参数，如指向解析后地址的迭代器 `it` 或用于保存接收到的数据的缓冲区 `buffer` 。

开始执行后，该应用将创建一个类型为 `boost::asio::ip::tcp::resolver::query` 的对象 `query` ，表示一个查询，其中含有名字 www.highscore.de 以及互联网常用的端口80。 这个查询被传递给 `async_resolve()` 方法以解析该名字。 最后， `main()` 只要调用 I/O 服务的 `run()` 方法，将控制交给操作系统进行异步操作即可。

当域名解析的过程完成后， `resolve_handler()` 被调用，检查域名是否能被解析。 如果解析成功，则存有错误条件的对象 `ec` 被设为0。 只有在这种情况下，才会相应地访问 socket 以创建连接。 服务器的地址是通过类型为 `boost::asio::ip::tcp::resolver::iterator` 的第二个参数来提供的。

调用了 `async_connect()` 方法之后， `connect_handler()` 会被自动调用。 在该句柄的内部，会访问 `ec` 对象以检查连接是否已建立。 如果连接是有效的，则对相应的 socket 调用 `async_read_some()` 方法，启动读数据操作。 为了保存接收到的数据，要提供一个缓冲区作为第一个参数。 在以上例子中，缓冲区的类型是 `boost::array` ，它来自 Boost C++ 库 Array，定义于 `boost/array.hpp` .

每当有一个或多个字节被接收并保存至缓冲区时， `read_handler()` 函数就会被调用。 准确的字节数通过 `std::size_t` 类型的参数 `bytes_transferred` 给出。 同样的规则，该句柄应该首先看看参数 `ec` 以检查有没有接收错误。 如果是成功接收，则将数据写出至标准输出流。

请留意， `read_handler()` 在将数据写出至 `std::cout` 之后，会再次调用 `async_read_some()` 方法。 这是必需的，因为无法保证仅在一次异步操作中就可以接收到整个网页。 `async_read_some()` 和 `read_handler()` 的交替调用只有当连接被破坏时才中止，如当 web 服务器已经传送完整个网页时。 这种情况

下，在 `read_handler()` 内部将报告一个错误，以防止进一步将数据输出至标准输出流，以及进一步对该 socket 调用 `async_read()` 方法。这时该例程将停止，因为没有更多的异步操作了。

上个例子是用来取出 www.highscore.de 的网页的，而下一个例子则示范了一个简单的 web 服务器。其主要差别在于，这个应用不会连接至其它PC，而是等待连接。

```
#include <boost/asio.hpp>
#include <string>

boost::asio::io_service io_service;
boost::asio::ip::tcp::endpoint endpoint(boost::asio::ip::tcp::v4(),
boost::asio::ip::tcp::acceptor acceptor(io_service, endpoint);
boost::asio::ip::tcp::socket sock(io_service);
std::string data = "HTTP/1.1 200 OK\r\nContent-Length: 13\r\n\r\nHe

void write_handler(const boost::system::error_code &ec, std::size_t
{
}

void accept_handler(const boost::system::error_code &ec)
{
  if (!ec)
  {
    boost::asio::async_write(sock, boost::asio::buffer(data), write
  }
}

int main()
{
  acceptor.listen();
  acceptor.async_accept(sock, accept_handler);
  io_service.run();
}
```

- 下载源代码

类型为 `boost::asio::ip::tcp::acceptor` 的 I/O 对象 `acceptor` - 被初始化为指定的协议和端口号 - 用于等待从其它PC传入的连接。初始化工作是通过 `endpoint` 对象完成的，该对象的类型为 `boost::asio::ip::tcp::endpoint`，将本例子中的接收器配置为使用端口80来等待 IP v4 的传入连接，这是 WWW 通常所使用的端口和协议。

接收器初始化完成后，`main()` 首先调用 `listen()` 方法将接收器置于接收状态，然后再用 `async_accept()` 方法等待初始连接。用于发送和接收数据的 socket 被作为第一个参数传递。

当一个PC试图建立一个连接时， `accept_handler()` 被自动调用。 如果该连接请求成功，就执行自由函数 `boost::asio::async_write()` 来通过 socket 发送保存在 `data` 中的信息。 `boost::asio::ip::tcp::socket` 还有一个名为 `async_write_some()` 的方法也可以发送数据；不过它会在发送了至少一个字节之后调用相关联的句柄。 该句柄需要计算还剩余多少字节，并反复调用 `async_write_some()` 直至所有字节发送完毕。 而使用 `boost::asio::async_write()` 可以避免这些，因为这个异步操作仅在缓冲区的所有字节都被发送后才结束。

在这个例子中，当所有数据发送完毕，空函数 `write_handler()` 将被调用。 由于所有异步操作都已完成，所以应用程序终止。 与其它PC的连接也被相应关闭。

## 7.5. 开发 Boost.Asio 扩展

虽然 Boost.Asio 主要是支持网络功能的，但是加入其它 I/O 对象以执行其它的异步操作也非常容易。 本节将介绍 Boost.Asio 扩展的一个总体布局。 虽然这不是必须的，但它为其它扩展提供了一个可行的框架作为起点。

要向 Boost.Asio 中增加新的异步操作，需要实现以下三个类：

- 一个派生自 `boost::asio::basic_io_object` 的类，以表示新的 I/O 对象。使用这个新的 Boost.Asio 扩展的开发者将只会看到这个 I/O 对象。
- 一个派生自 `boost::asio::io_service::service` 的类，表示一个服务，它被注册为 I/O 服务，可以从 I/O 对象访问它。 服务与 I/O 对象之间的区别是很重要的，因为在任意给定的时间点，每个 I/O 服务只能有一个服务实例，而一个服务可以被多个 I/O 对象访问。
- 一个不派生自任何其它类的类，表示该服务的具体实现。 由于在任意给定的时间点每个 I/O 服务只能有一个服务实例，所以服务会为每个 I/O 对象创建一个其具体实现的实例。 该实例管理与相应 I/O 对象有关的内部数据。

本节中开发的 Boost.Asio 扩展并不仅仅提供一个框架，而是模拟一个可用的 `boost::asio::deadline_timer` 对象。 它与原来的 `boost::asio::deadline_timer` 的区别在于，计时器的时长是作为参数传递给 `wait()` 或 `async_wait()` 方法的，而不是传给构造函数。

```
#include <boost/asio.hpp>
#include <cstddef>

template <typename Service>
class basic_timer
  : public boost::asio::basic_io_object<Service>
{
  public:
    explicit basic_timer(boost::asio::io_service &io_service)
      : boost::asio::basic_io_object<Service>(io_service)
    {
    }

    void wait(std::size_t seconds)
    {
      return this->service.wait(this->implementation, seconds);
    }

    template <typename Handler>
    void async_wait(std::size_t seconds, Handler handler)
    {
      this->service.async_wait(this->implementation, seconds, handl
    }
};
```

- 下载源代码

每个 I/O 对象通常被实现为一个模板类，要求以一个服务来实例化 - 通常就是那个特定为此 I/O 对象开发的服务。 当一个 I/O 对象被实例化时，该服务会通过父类 `boost::asio::basic_io_object` 自动注册为 I/O 服务，除非它之前已经注册。这样可确保任何 I/O 对象所使用的服务只会每个 I/O 服务只注册一次。

在 I/O 对象的内部，可以通过 `service` 引用来访问相应的服务，通常的访问就是将方法调用前转至该服务。 由于服务需要为每一个 I/O 对象保存数据，所以要为每一个使用该服务的 I/O 对象自动创建一个实例。 这还是在父类 `boost::asio::basic_io_object` 的帮助下实现的。 实际的服务实现被作为一个参数传递给任一方法调用，使得服务可以知道是哪个 I/O 对象启动了这次调用。 服务的具体实现是通过 `implementation` 属性来访问的。

一般一上谕，I/O 对象是相对简单的：服务的安装以及服务实现的创建都是由父类 `boost::asio::basic_io_object` 来完成的，方法调用则只是前转至相应的服务；以 I/O 对象的实际服务实现作为参数即可。

```
#include <boost/asio.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <boost/scoped_ptr.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
```

```
#include <boost/system/error_code.hpp>

template <typename TimerImplementation = timer_impl>
class basic_timer_service
  : public boost::asio::io_service::service
{
  public:
    static boost::asio::io_service::id id;

    explicit basic_timer_service(boost::asio::io_service &io_servic
      : boost::asio::io_service::service(io_service),
      async_work_(new boost::asio::io_service::work(async_io_servic
      async_thread_(boost::bind(&boost::asio::io_service::run, &asy
    {
    }

    ~basic_timer_service()
    {
      async_work_.reset();
      async_io_service_.stop();
      async_thread_.join();
    }

    typedef boost::shared_ptr<TimerImplementation> implementation_t

    void construct(implementation_type &impl)
    {
      impl.reset(new TimerImplementation());
    }

    void destroy(implementation_type &impl)
    {
      impl->destroy();
      impl.reset();
    }

    void wait(implementation_type &impl, std::size_t seconds)
    {
      boost::system::error_code ec;
      impl->wait(seconds, ec);
      boost::asio::detail::throw_error(ec);
    }

    template <typename Handler>
    class wait_operation
    {
      public:
        wait_operation(implementation_type &impl, boost::asio::io_s
          : impl_(impl),
          io_service_(io_service),
          work_(io_service),
          seconds_(seconds),
          handler_(handler)
```

```
        {
        }

        void operator()() const
        {
          implementation_type impl = impl_.lock();
          if (impl)
          {
              boost::system::error_code ec;
              impl->wait(seconds_, ec);
              this->io_service_.post(boost::asio::detail::bind_hand
          }
          else
          {
              this->io_service_.post(boost::asio::detail::bind_hand
          }
      }

      private:
        boost::weak_ptr<TimerImplementation> impl_;
        boost::asio::io_service &io_service_;
        boost::asio::io_service::work work_;
        std::size_t seconds_;
        Handler handler_;
    };

    template <typename Handler>
    void async_wait(implementation_type &impl, std::size_t seconds,
    {
      this->async_io_service_.post(wait_operation<Handler>(impl, th
    }

  private:
    void shutdown_service()
    {
    }

    boost::asio::io_service async_io_service_;
    boost::scoped_ptr<boost::asio::io_service::work> async_work_;
    boost::thread async_thread_;
};

template <typename TimerImplementation>
boost::asio::io_service::id basic_timer_service<TimerImplementation
```

- 下载源代码

为了与 Boost.Asio 集成，一个服务必须符合几个要求：

- 它必须派生自 `boost::asio::io_service::service` 。 构造函数必须接受一个指向 I/O 服务的引用，该 I/O 服务会被相应地传给 `boost::asio::io_service::service` 的构造函数。

- 任何服务都必须包含一个类型为 `boost::asio::io_service::id` 的静态公有属性 `id` 。在 I/O 服务的内部是用该属性来识别服务的。

- 必须定义两个名为 `construct()` 和 `destruct()` 的公有方法，均要求一个类型为 `implementation_type` 的参数。 `implementation_type` 通常是该服务的具体实现的类型定义。 正如上面例子所示，在 `construct()` 中可以很容易地使用一个 `boost::shared_ptr` 对象来初始化一个服务实现，以及在 `destruct()` 中相应地析构它。 由于这两个方法都会在一个 I/O 对象被创建或销毁时自动被调用，所以一个服务可以分别使用 `construct()` 和 `destruct()` 为每个 I/O 对象创建和销毁服务实现。

- 必须定义一个名为 `shutdown_service()` 的方法；不过它可以是私有的。 对于一般的 Boost.Asio 扩展来说，它通常是一个空方法。 只有与 Boost.Asio 集成得非常紧密的服务才会使用它。 但是这个方法必须要有，这样扩展才能编译成功。

为了将方法调用前转至相应的服务，必须为相应的 I/O 对象定义要前转的方法。 这些方法通常具有与 I/O 对象中的方法相似的名字，如上例中的 `wait()` 和 `async_wait()` 。 同步方法，如 `wait()` ，只是访问该服务的具体实现去调用一个阻塞式的方法，而异步方法，如 `async_wait()` ，则是在一个线程中调用这个阻塞式方法。

在线程的协助下使用异步操作，通常是通过访问一个新的 I/O 服务来完成的。 上述例子中包含了一个名为 `async_io_service_` 的属性，其类型为 `boost::asio::io_service` 。 这个 I/O 服务的 `run()` 方法是在它自己的线程中启动的，而它的线程是在该服务的构造函数内部由类型为 `boost::thread` 的 `async_thread_` 创建的。 第三个属性 `async_work_` 的类型为 `boost::scoped_ptr<boost::asio::io_service::work>` ，用于避免 `run()` 方法立即返回。 否则，这可能会发生，因为已没有其它的异步操作在创建。 创建一个类型为 `boost::asio::io_service::work` 的对象并将它绑定至该 I/O 服务，这个动作也是发生在该服务的构造函数中，可以防止 `run()` 方法立即返回。

一个服务也可以无需访问它自身的 I/O 服务来实现 - 单线程就足够的。 为新增的线程使用一个新的 I/O 服务的原因是，这样更简单： 线程间可以用 I/O 服务来非常容易地相互通信。 在这个例子中， `async_wait()` 创建了一个类型为 `wait_operation` 的函数对象，并通过 `post()` 方法将它传递给内部的 I/O 服务。 然后，在用于执行这个内部 I/O 服务的 `run()` 方法的线程内，调用该函数对象的重载 `operator()()` 。 `post()` 提供了一个简单的方法，在另一个线程中执行一个函数对象。

`wait_operation` 的重载 `operator()()` 操作符基本上就是执行了和 `wait()` 方法相同的工作：调用服务实现中的阻塞式 `wait()` 方法。 但是，有可能这个 I/O 对象以及它的服务实现在这个线程执行 `operator()()` 操作符期间被销毁。

如果服务实现是在 `destruct()` 中销毁的，则 `operator()()` 操作符将不能再访问它。 这种情形是通过使用一个弱指针来防止的，从第一章中我们知道：如果在调用 `lock()` 时服务实现仍然存在，则弱指针 `impl_` 返回它的一个共享指针，否则它将返回0。 在这种情况下， `operator()()` 不会访问这个服务实现，而是以一个 `boost::asio::error::operation_aborted` 错误来调用句柄。

```
#include <boost/system/error_code.hpp>
#include <cstddef>
#include <windows.h>

class timer_impl
{
  public:
    timer_impl()
      : handle_(CreateEvent(NULL, FALSE, FALSE, NULL))
    {
    }

    ~timer_impl()
    {
      CloseHandle(handle_);
    }

    void destroy()
    {
      SetEvent(handle_);
    }

    void wait(std::size_t seconds, boost::system::error_code &ec)
    {
      DWORD res = WaitForSingleObject(handle_, seconds * 1000);
      if (res == WAIT_OBJECT_0)
        ec = boost::asio::error::operation_aborted;
      else
        ec = boost::system::error_code();
    }

private:
    HANDLE handle_;
};
```

- 下载源代码

服务实现 `timer_impl` 使用了 Windows API 函数，只能在 Windows 中编译和使用。 这个例子的目的只是为了说明一种潜在的实现。

`timer_impl` 提供两个基本方法： `wait()` 用于等待数秒。 `destroy()` 则用于取消一个等待操作，这是必须要有的，因为对于异步操作来说， `wait()` 方法是在其自身的线程中调用的。 如果 I/O 对象及其服务实现被销毁，那么阻塞式的

`wait()` 方法就要尽使用 `destroy()` 来取消。

这个 Boost.Asio 扩展可以如下使用。

```
#include <boost/asio.hpp>
#include <iostream>
#include "basic_timer.hpp"
#include "timer_impl.hpp"
#include "basic_timer_service.hpp"

void wait_handler(const boost::system::error_code &ec)
{
  std::cout << "5 s." << std::endl;
}

typedef basic_timer<basic_timer_service<> > timer;

int main()
{
  boost::asio::io_service io_service;
  timer t(io_service);
  t.async_wait(5, wait_handler);
  io_service.run();
}
```

- 下载源代码

与本章开始的例子相比，这个 Boost.Asio 扩展的用法类似于 `boost::asio::deadline_timer` 。 在实践上，应该优先使用 `boost::asio::deadline_timer` ，因为它已经集成在 Boost.Asio 中了。 这个扩展的唯一目的就是示范一下 Boost.Asio 是如何扩展新的异步操作的。

目录监视器(Directory Monitor) 是现实中的一个 Boost.Asio 扩展，它提供了一个可以监视目录的 I/O 对象。 如果被监视目录中的某个文件被创建、修改或是删除，就会相应地调用一个句柄。 当前的版本支持 Windows 和 Linux (内核版本 2.6.13 或以上)。

## 7.6. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 修改 第 7.4 节 “网络编程” 中的服务器程序，不在一次请求后即终止，而是可以处理任意多次请求。

2. 扩展 第 7.4 节 “网络编程” 中的客户端程序，即时在所接收到的HTML代码中分析某个URL。 如果找到，则同时下载相应的资源。 对于本练习，只使用第一个URL。 理想情况下，网站及其资源应被保存在两个文件中而不是同时写出至标准输出流。

3. 创建一个客户端/服务器应用，在两台PC间传送文件。 当服务器端启动后，它应该显示所有本地接口的IP地址并等待客户端连接。 客户端则应将服务器端的某一个IP地址以及某个本地文件的文件名作为命令行参数。 客户端应将该文件传送给服务器，后者则相应地保存它。 在传送过程中，客户端应向用户提供一些进度的可视显示。

# 第 8 章 进程间通讯

## 目录





## 8.1. 概述

进程间通讯描述的是同一台计算机的不同应用程序之间的数据交换机制。 但不包括网络通讯方式。 如果需要经由网络，在彼此运行在不同计算机上的应用程序之间交换数据，请看第 7 章 异步输入输出，该章讲述了 Boost.Asio 库。

本章展示了 Boost.Interprocess 库，它包括众多的类，这些类提供了操作系统相关的进程间通讯接口的抽象层。 虽然不同操作系统的进程间通讯概念非常相近，但接口的变化却很大。 Boost.Interprocess 库使通过C++使用这些功能成为可能。

虽然 Boost.Asio 也可以用来在同一台计算机的应用程序间交换数据，但是使用 Boost.Interprocess 库通常性能更好。 Boost.Interprocess 库实际上是使用操作系统的功能优化了同一台计算机的应用程序之间数据交换，所以它应该是任何不需要网络时应用程序间数据交换的首选。

## 8.2. 共享内存

共享内存通常是进程间通讯最快的形式。 它提供一块在应用程序间共享的内存区域。 一个应用能够在另一个应用读取数据时写数据。

这样一块内存区用 Boost.Interprocess 的 `boost::interprocess::shared_memory_object` 类表示。 为使用这个类，需要包含 `boost/interprocess/shared_memory_object.hpp` 头文件。

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object shdmem(boost::interproc
  shdmem.truncate(1024);
  std::cout << shdmem.get_name() << std::endl;
  boost::interprocess::offset_t size;
  if (shdmem.get_size(size))
    std::cout << size << std::endl;
}
```

- 下载源代码

`boost::interprocess::shared_memory_object` 的构造函数需要三个参数。第一个参数指定共享内存是要创建或打开。上面的例子实际上是指定了两种方式：用 `boost::interprocess::open_or_create` 作为参数，共享内存如果存在就将其打开，否则创建之。

假设之前已经创建了共享内存，现打开前面已经创建的共享内存。为了唯一标识一块共享内存，需要为其指定一个名称，传递给 `boost::interprocess::shared_memory_object` 构造函数的第二个参数指定了这个名称。

第三个，也就是最后一个参数指示应用程序如何访问共享内存。例子应用程序能够读写共享内存，这是因为最后的一个参数是 `boost::interprocess::read_write`。

在创建一个 `boost::interprocess::shared_memory_object` 类型的对象后，相应的共享内存就在操作系统中建立了。可是此共享内存区域的大小被初始化为0. 为了使用这块区域，需要调用 `truncate()` 函数，以字节为单位传递请求的共享内存的大小。对于上面的例子，共享内存提供了1,024字节的空间。

请注意，`truncate()` 函数只能在共享内存以 `boost::interprocess::read_write` 方式打开时调用。如果不是以此方式打开，将抛出 `boost::interprocess::interprocess_exception` 异常。

为了调整共享内存的大小，`truncate()` 函数可以被重复调用。

在创建共享内存后，`get_name()` 和 `get_size()` 函数可以分别用来查询共享内存的名称和大小。

由于共享内存被用于应用程序之间交换数据，所以每个应用程序需要映射共享内存到自己的地址空间上，这是通过 `boost::interprocess::mapped_region` 类实现的。

你也许有些奇怪，为了访问共享内存，要使用两个类。 是的， `boost::interprocess::mapped_region` 还能映射不同的对象到具体应用的地址空间。 如 Boost.Interprocess 提供 `boost::interprocess::file_mapping` 类实际上代表特定文件的共享内存。 所以 `boost::interprocess::file_mapping` 类型的对象对应一个文件。 向这个对象写入的数据将自动保存关联的物理文件上。 由于 `boost::interprocess::file_mapping` 不必加载整个文件，但却可以使用 `boost::interprocess::mapped_region` 将任意部分映射到地址空间，所以就能处理几个GB的文件，而这个文件在32位系统上是不能全部加载到内存上的。

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object shdmem(boost::interproc
  shdmem.truncate(1024);
  boost::interprocess::mapped_region region(shdmem, boost::interpro
  std::cout << std::hex << "0x" << region.get_address() << std::end
  std::cout << std::dec << region.get_size() << std::endl;
  boost::interprocess::mapped_region region2(shdmem, boost::interpr
  std::cout << std::hex << "0x" << region2.get_address() << std::en
  std::cout << std::dec << region2.get_size() << std::endl;
}
```

- 下载源代码

为了使用 `boost::interprocess::mapped_region` 类，需要包含 `boost/interprocess/mapped_region.hpp` 头文件。 `boost::interprocess::mapped_region` 的构造函数的第一个参数必须是 `boost::interprocess::shared_memory_object` 类型的对象。 第二个参数指示此内存区域对应用程序来说，是只读或是可写的。

上面的例子创建了两个 `boost::interprocess::mapped_region` 类型的对象。 名为"Highscore"的共享内存，被映射到进程的地址空间两次。 通过 `get_address()` 和 `get_size()` 这两个函数获得共享内存的地址和大小写到标准标准输出流中。 在这两种情况下， `get_size()` 的返回值都是 `1024` ，而 `get_address()` 的返回值是不同的。

下面的例子使用共享内存写入并读取一个数字。

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object shdmem(boost::interproc
  shdmem.truncate(1024);
  boost::interprocess::mapped_region region(shdmem, boost::interpro
  int *i1 = static_cast<int*>(region.get_address());
  *i1 = 99;
  boost::interprocess::mapped_region region2(shdmem, boost::interpr
  int *i2 = static_cast<int*>(region2.get_address());
  std::cout << *i2 << std::endl;
}
```

- 下载源代码

通过变量 `region` ,数值 99 被写到共享内存的开始处。 然后变量 `region2` 访问共享内存的同一个位置，并将数值写入到标准输出流中。 正如前面例子的 `get_address()` 函数的返回值所见，虽然变量 `region` 和 `region2` 表示的是该进程内不同的内存区域，但由于两个内存区域底层实际访问的是同一块共享内存，所以程序打印出 `99` 。

通常，不会在同一个应用程序内使用多个 `boost::interprocess::mapped_region` 访问同一块共享内存。 实际上在同一个应用程序内将同一个共享内存映射到不同的内存区域上没有多大的意义，上面的例子只用于说明的目的。

为了删除指定的共享内存， `boost::interprocess::shared_memory_object` 对象提供了静态的 `remove()` 函数，此函数带有一个要被删除的共享内存名称的参数。

Boost.Interprocess 类的RAII概念支持，明显来自关于智能指针的章节，并使用了另外的一个类名称 `boost::interprocess::remove_shared_memory_on_destroy` 。 它的构造函数需要一个已经存在的共享内存的名称。 如果这个类的对象被销毁了，那么在析构函数中会自动删除共享内存的容器。

请注意构造函数并不创建或打开共享内存，所以，这个类并不是典型RAII概念的代表。

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <iostream>

int main()
{
  bool removed = boost::interprocess::shared_memory_object::remove(
  std::cout << removed << std::endl;
}
```

- 下载源代码

如果 `remove()` 没有被调用, 那么，即使进程终止，共享内存还会一直存在，而不论共享内存的删除是否依赖底层操作系统。 多数Unix操作系统，包括Linux，一旦系统重新启动，都会自动删除共享内存，在 Windows 或 Mac OS X 上， `remove()` 必须调用，这两种系统实际上将共享内存存储在持久化的文件上，此文件在系统重启后还是存在的。

Windows 提供了一种特别的共享内存，它可以在最后一个使用它的应用程序终止后自动删除。 为了使用它，提供了 `boost::interprocess::windows_shared_memory` 类，定义在 `boost/interprocess/windows_shared_memory.hpp` 文件中。

```
#include <boost/interprocess/windows_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

int main()
{
  boost::interprocess::windows_shared_memory shdmem(boost::interpro
  boost::interprocess::mapped_region region(shdmem, boost::interpro
  int *i1 = static_cast<int*>(region.get_address());
  *i1 = 99;
  boost::interprocess::mapped_region region2(shdmem, boost::interpr
  int *i2 = static_cast<int*>(region2.get_address());
  std::cout << *i2 << std::endl;
}
```

- 下载源代码

请注意， `boost::interprocess::windows_shared_memory` 类没有提供 `truncate()` 函数，而是在构造函数的第四个参数传递共享内存的大小。

即使 `boost::interprocess::windows_shared_memory` 类是不可移植的，且只能用于Windows系统，但使用这种特别的共享内存在不同应用之间交换数据，它还是非常有用的。

## 8.3. 托管共享内存

上一节介绍了用来创建和管理共享的 `boost::interprocess::shared_memory_object` 类。 实际上，由于这个类需要按单个字节的方式读写共享内存，所以这个类几乎不用。 概念上来讲，C++改善了类对象的创建并隐藏了它们存储在内存中哪里，是怎们存储的这些细节。

Boost.Interprocess 提供了一个名为“托管共享内存”的概念，通过定义在 `boost/interprocess/managed_shared_memory.hpp` 文件中的 `boost::interprocess::managed_shared_memory` 类提供。 这个类的目的是，对于需要分配到共享内存上的对象，它能够以内存申请的方式初始化，并使其自动为使用同一个共享内存的其他应用程序可用。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object::remove("Highscore");
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  int *i = managed_shm.construct<int>("Integer")(99);
  std::cout << *i << std::endl;
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer")
  if (p.first)
    std::cout << *p.first << std::endl;
}
```

- 下载源代码

上面的例子打开名为 "Highscore" 大小为1,024 字节的共享内存，如果它不存在，它会被自动地创建。

在常规的共享内存中，为了读写数据，单个字节被直接访问，托管共享内存使用诸如 `construct()` 函数，此函数要求一个数据类型作为模板参数，此例中声明的是 `int` 类型，函数缺省要求一个名称来表示在共享内存中创建的对象。 此例中使用的名称是 "Integer"。

由于 `construct()` 函数返回一个代理对象，为了初始化创建的对象，可以传递参数给此函数。 语法看上去像调用一个构造函数。 这就确保了对象不仅能在共享内存上创建，还能够按需要的方式初始化它。

为了访问托管共享内存上的一个特定对象，用 `find()` 函数。 通过传递要查找对象的名称，返回或者是一个指向这个特定对象的指针，或者是0表示给定名称的对象没有找到。

正如前面例子中所见， `find()` 实际返回的是 `std::pair` 类型的对象， `first` 属性提供的是指向对象的指针，那么 `second` 属性提供的是什么呢？

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object::remove("Highscore");
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  int *i = managed_shm.construct<int>("Integer")[10](99);
  std::cout << *i << std::endl;
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer")
  if (p.first)
  {
    std::cout << *p.first << std::endl;
    std::cout << p.second << std::endl;
  }
}
```

- 下载源代码

这次，通过在 `construct()` 函数后面给以用方括号括住的数字10，创建了一个10个元素的 `int` 类型的数组。 将 `second` 属性写到标准输出流，同样是这个数字 `10` 。 使用这个属性， `find()` 函数返回的对象能够区分是单个对象还是数组对象。 对于前者， `second` 的值是1，而对于后者，它的值是数组元素的个数。

请注意数组中的所有元素被初始化为数值99。 不可能每个元素初始化为不同的值。

如果给定名称的对象已经在托管的共享内存中存在，那么 `construct()` 将会失败。 在这种情况下， `construct()` 返回值是0。 如果存在的对象即使存在也可以被重用， `find_or_construct()` 函数可以调用，此函数返回一个指向它找到的对象的指针。 此时没有初始化动作发生。

其他可以导致 `construct()` 失败的情况如下例所示。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::interprocess::shared_memory_object::remove("Highscore");
    boost::interprocess::managed_shared_memory managed_shm(boost::i
    int *i = managed_shm.construct<int>("Integer")[4096](99);
  }
  catch (boost::interprocess::bad_alloc &ex)
  {
    std::cerr << ex.what() << std::endl;
  }
}
```

- 下载源代码

应用程序尝试创建一个 `int` 类型的，包含4,096个元素的数组。 然而，共享内存只有1,024 字节，于是由于共享内存不能提供请求的内存，而抛出 `boost::interprocess::bad_alloc` 类型的异常。

一旦对象已经在共享内存中创建，它们可以用 `destroy()` 函数删除。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object::remove("Highscore");
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  int *i = managed_shm.find_or_construct<int>("Integer")(99);
  std::cout << *i << std::endl;
  managed_shm.destroy<int>("Integer");
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer")
  std::cout << p.first << std::endl;
}
```

- 下载源代码

由于它是一个参数的，要被删除对象的名称传递给 `destroy()` 函数。 如果需要，可以检查此函数的 `bool` 类型的返回值，以确定是否给定的对象被找到并成功删除。 由于对象如果被找到总是被删除，所以返回值 `false` 表示给定名称的对象没有找到。

除了 `destroy()` 函数，还提供了另外一个函数 `destroy_ptr()` ，它能够传递一个托管共享内存中的对象的指针，它也能用来删除数组。

由于托管内存很容易用来存储在不同应用程序之间共享的对象，那么很自然就会使用到来自C++标准模板库的容器了。 这些容器需要用 `new` 这种方式来分配各自需要的内存。 为了在托管共享内存上使用这些容器，这就需要更加仔细地在共享内存上分配内存。

可惜的是，许多C++标准模板库的实现并不太灵活，不能够提供 Boost.Interprocess 使用 `std::string` 或 `std::list` 的容器。 移植到 Microsoft Visual Studio 2008 的标准模板库实现就是一个例子。

为了允许开发人员可以使用这些有名的来自C++标准的容器，Boost.Interprocess 在命名空间 `boost::interprocess` 下，提供了它们的更灵活的实现方式。 如， `boost::interprocess::string` 的行为实际上对应的是 `std::string` ，优点是它的对象能够安全地存储在托管共享内存上。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <iostream>

int main()
{
  boost::interprocess::shared_memory_object::remove("Highscore");
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  typedef boost::interprocess::allocator<char, boost::interprocess
  typedef boost::interprocess::basic_string<char, std::char_traits<
  string *s = managed_shm.find_or_construct<string>("String")("Hel
  s->insert(5, ", world");
  std::cout << *s << std::endl;
}
```

- 下载源代码

为了创建在托管共享内存上申请内存的字符串，必须为 Boost.Interprocess 提供的另外一个分配器定义对应的数据类型，而不是使用C++标准提供的缺省分配器。

为了这个目的，Boost.Interprocess 在 `boost/interprocess/allocators/allocator.hpp` 文件中提供了 `boost::interprocess::allocator` 类的定义。 使用这个类，可以创建一个分配器，此分配器的内部使用的是“托管共享内存段管理器”。 段管理器负责管理位于托管共享内存之内的内存。 使用新建的分配器，与 string 相应的数据类型被定义了。 如上面所示，它使用 `boost::interprocess::basic_string` 而不是 `std::basic_string` 。 上面例子中的新数据类型简单地命名为 `string` ，它是基于 `boost::interprocess::basic_string` 并经过分配器访问段管理器。 为了让通过 `find_or_construct()` 创建的 `string` 特定实例，知道哪个段管理器应该被访问，相应的段管理器的指针传递给构造函数的第二个参数。

与 `boost::interprocess::string` 一起, Boost.Interprocess 还提供了许多其他 C++标准中已知的容器。 如, `boost::interprocess::vector` 和 `boost::interprocess::map` , 分别定义在 `boost/interprocess/containers/vector.hpp` 和 `boost/interprocess/containers/map.hpp` 文件中

无论何时同一个托管共享内存被不同的应用程序访问，诸如创建，查找和销毁对象的操作是自动同步的。 如果两个应用程序尝试在托管共享内存上创建不同名称的对象，访问相应地被串行化了。 为了立刻执行多个操作而不被其他应用程序的操作打断，可以使用 `atomic_func()` 函数。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/bind.hpp>
#include <iostream>

void construct_objects(boost::interprocess::managed_shared_memory &
{
  managed_shm.construct<int>("Integer")(99);
  managed_shm.construct<float>("Float")(3.14);
}

int main()
{
  boost::interprocess::shared_memory_object::remove("Highscore");
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  managed_shm.atomic_func(boost::bind(construct_objects, boost::ref
  std::cout << *managed_shm.find<int>("Integer").first << std::endl
  std::cout << *managed_shm.find<float>("Float").first << std::endl
}
```

- 下载源代码

`atomic_func()` 需要一个无参数，无返回值的函数作为它的参数。 被传递的函数将以以一种确保排他访问托管共享内存的方式被调用，但仅限于对象的创建，查找和销毁操作。 如果另一个应用程序有一个指向托管内存中对象的指针，它还是可以使用这个指针修改该对象的。

Boost.Interprocess 也可以用来同步对象的访问。 由于 Boost.Interprocess 不知道在任意一个时间点谁可以访问某个对象，所以同步需要明确的状态标志，下一节介绍这些类提供的同步方式。

## 8.4. 同步

Boost.Interprocess 允许多个应用程序并发使用共享内存。 由于共享内存被定义为在应用程序之间"共享"，所以 Boost.Interprocess 需要支持一些同步方式。

当考虑到同步的时候，Boost.Thread 当然浮现在脑海里。 正如在 第 6 章 多线程 所见，Boost.Thread 确实提供了各种概念，如互斥对象和条件变量来同步线程。 可惜的是，这些类只能用来同步同一个应用程序内的线程，它们不支持同步不同的应用程序。 由于二者面临的问题相同，所以在概念上没有什么差别。

当诸如互斥对象和条件变量等同步对象位于一个多线程的应用程序的同一地址空间内时，当然它们对于所有线程都是可以访问的，而在共享内存方面的问题是不同的应用程序需要在彼此之间正确地共享这些对象。 例如，如果一个应用程序创建一个互斥对象，它有时候需要从另外一个应用程序访问此对象。

Boost.Interprocess 提供了两种同步对象，匿名对象被直接存储在共享内存上，这使得他们自动对所有应用程序可用。 命名对象由操作系统管理，所以它们不存储在共享内存上，它们可以被应用程序通过名称访问。

接下来的例子通过 `boost::interprocess::named_mutex` 创建并使用一个命名互斥对象，此类定义在 `boost/interprocess/sync/named_mutex.hpp` 文件中。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <iostream>

int main()
{
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  int *i = managed_shm.find_or_construct<int>("Integer")();
  boost::interprocess::named_mutex named_mtx(boost::interprocess::c
  named_mtx.lock();
  ++(*i);
  std::cout << *i << std::endl;
  named_mtx.unlock();
}
```

- 下载源代码

除了一个参数用来指定互斥对象是被创建或者打开之外， `boost::interprocess::named_mutex` 的构造函数还需要一个名称参数。 每个知道此名称的应用程序能够访问这同一个对象。 为了获得对位于共享内存中数据的访问，应用程序需要通过调用 `lock()` 函数获得互斥对象的拥有关系。 由于互斥对象在任意时刻只能被一个应用程序拥有，其他应用程序需要等待，直到互斥对象被第一个应用程序使用 `lock()` 函数释放。 一旦应用程序获得互斥对象的所有权，它可以获得互斥对象保护的资源的排他访问。 在上面例子中，资源是 `int` 类的变量被递增并写到标准输出流中。

如果应用程序被启动多次，每个实例都会打印出和前一个值比较递增1的值。 感谢互斥对象，访问共享内存和变量本身在多个应用程序之间是同步的。

接下来的应用程序使用了定义在 `boost/interprocess/sync/interprocess_mutex.hpp` 文件中的 `boost::interprocess::interprocess_mutex` 类的匿名对象。 为了可以被所有应用程序访问，它存储在共享内存中。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <iostream>

int main()
{
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  int *i = managed_shm.find_or_construct<int>("Integer")();
  boost::interprocess::interprocess_mutex *mtx = managed_shm.find_c
  mtx->lock();
  ++(*i);
  std::cout << *i << std::endl;
  mtx->unlock();
}
```

- 下载源代码

这个应用程序的行为确实和前一个有点像。 唯一的不同是这次互斥对象通过用 `boost::interprocess::managed_shared_memory` 类的 `construct()` 或 `find_or_construct()` 函数被直接存储在共享内存中。

除了 `lock()` 函数， `boost::interprocess::named_mutex` 和 `boost::interprocess::interprocess_mutex` 还提供了 `try_lock()` 和 `timed_lock()` 函数。 它们的行为和Boost.Thread提供的互斥对象相对应。

在需要递归互斥对象的时候，Boost.Interprocess 提供 `boost::interprocess::named_recursive_mutex` 和 `boost::interprocess::interprocess_mutex` 两个对象可供使用。

在互斥对象保证共享资源的排他访问的时候，条件变量控制了在什么时候，谁必须具有排他访问权。 一般来讲，Boost.Interprocess 和 Boost.Thread 提供的条件变量工作方式相同。 它们有非常相似的接口，使熟悉 Boost.Thread 的用户在使用 Boost.Interprocess 的这些条件变量时立刻有一种自在的感觉。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <boost/interprocess/sync/named_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>

int main()
{
  boost::interprocess::managed_shared_memory managed_shm(boost::int
  int *i = managed_shm.find_or_construct<int>("Integer")(0);
  boost::interprocess::named_mutex named_mtx(boost::interprocess::c
  boost::interprocess::named_condition named_cnd(boost::interproces
  boost::interprocess::scoped_lock<boost::interprocess::named_mute
  while (*i < 10)
  {
    if (*i % 2 == 0)
    {
      ++(*i);
      named_cnd.notify_all();
      named_cnd.wait(lock);
    }
    else
    {
      std::cout << *i << std::endl;
      ++(*i);
      named_cnd.notify_all();
      named_cnd.wait(lock);
    }
  }
  named_cnd.notify_all();
  boost::interprocess::shared_memory_object::remove("shm");
  boost::interprocess::named_mutex::remove("mtx");
  boost::interprocess::named_condition::remove("cnd");
}
```

- 下载源代码

例子中使用的条件变量的类型 `boost::interprocess::named_condition` ，定义在 `boost/interprocess/sync/named_condition.hpp` 文件中。 由于它是命名变量，所以它不需要存储在共享内存。

应用程序使用 `while` 循环递增一个存储在共享内存中的 `int` 类型变量而变量是在每个循环内重复递增，而它只在每两个循环时写出到标准输出中：写出的只能是奇数。

每次，在变量递增1之后，条件变量 `named_cnd` 的 `wait()` 函数被调用。 一个称作锁，在此例中是变量 `lock` 被传递给此函数。 这个锁和 Boost.Thread 中的锁含义相同：基于RAII概念的在构造函数中获得互斥对象的所有权,并在析构函数中释放它。

在 `while` 之前创建的锁因而在整个应用程序执行期间拥有互斥对象的所有权。可是，如果作为一个参数传递给 `wait()` 函数，它会被自动释放。

条件变量常常用来等待一个信号，此信号会指示等待现在到了。 同步是通过 `wait()` 和 `notify_all()` 函数控制的。 如果一个应用程序调用 `wait()` 函数，一直到对应的条件变量的 `notify_all()` 函数被调用，相应的互斥对象的所有权才会被被释放。

如果启动此程序，它看上去什么也没做：而只是变量在 `while` 循环内从0递增到1，然后应用程序使用 `wait()` 等待信号。 为了提供这个信号，应用程序需要再启动第二个实例。

应用程序的第二个实例将会在进入 `while` 循环之前，尝试获得同一个互斥对象的所有权。 这肯定是成功的，由于应用程序的第一个实例通过调用 `wait()` 释放了互斥对象的所有权。 因为变量已经递增了一次，第二个实例现在会执行 `if` 表达式的 `else` 分支，这使得在递增1之前将当前值写到标准输出流。

现在，第二个实例也调用了 `wait()` 函数，可是，在调用之前,它调用了 `notify_all()` 函数，这对于两个实例正确协作是非常重要的顺序。 第一个实例被通知并再次尝试获得互斥对象的所有权，虽然现在它还被第二个实例所拥有。 由于第二个实例在调用 `notify_all()` 之后调用了 `wait()`，这自动释放了所有权，第一个实例此时能够获得所有权。

两个实例交替地递增共享内存中的变量。 仅有一个实例将变量值写到标准输出流。 只要变量值到达10， `while` 循环结束。 为了让其他实例不必永远等待信号， `notify_all()` 函数在循环之后又被调用了一次。 在终止之前，共享内存，互斥对象和条件变量都被销毁。

就像有两种互斥对象，即必须存储在共享内存中匿名类型和命名类型，也存在两种类型的条件变量。 现在用匿名条件变量重写上面的例子。

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/interprocess_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::interprocess::managed_shared_memory managed_shm(boost::i
    int *i = managed_shm.find_or_construct<int>("Integer")(0);
    boost::interprocess::interprocess_mutex *mtx = managed_shm.find
    boost::interprocess::interprocess_condition *cnd = managed_shm.
    boost::interprocess::scoped_lock<boost::interprocess::interproc
    while (*i < 10)
    {
      if (*i % 2 == 0)
      {
        ++(*i);
        cnd->notify_all();
        cnd->wait(lock);
      }
      else
      {
        std::cout << *i << std::endl;
        ++(*i);
        cnd->notify_all();
        cnd->wait(lock);
      }
    }
    cnd->notify_all();
  }
  catch (...)
  {
  }
  boost::interprocess::shared_memory_object::remove("shm");
}
```

- 下载源代码

这个应用程序的工作完全和前一个例子一样，为了递增变量10次，因而也需要启动两个实例。 两个例子之间的差别很小。 与是否使用匿名或命名条件变量根本没有什么关系。

处理互斥对象和条件变量，Boost.Interprocess 还提供了叫做信号量和文件锁。 信号量的行为和条件变量相似，除了它不能区别两种状态，但它确是基于计数器的。 文件锁有些像互斥对象， 虽然它们不是关于内存的对象，但它们确是文件系统上关于文件的对象。

就像 Boost.Thread 能够区分不同的互斥对象和锁，Boost.Interprocess 也提供了几个互斥对象和锁。 例如，互斥对象不仅能被排他地拥有，也可以不排他地所有。 这在多个应用程序需要同时读而排他写的时候非常有用。 对于不同的互斥对象，可以使用不同的具有RAII概念的锁类。

请注意如果不使用匿名同步对象，那么名称应该是唯一的。 虽然互斥对象和条件变量是基于不同类的对象，但也不必总是认为操作系统独立的接口是由 Boost.Interprocess 区别对待的。 在Windows系统上，互斥对象和条件变量使用同样的系统函数。 如果这两种对象使用相同的名称，那么应用程序在Windows上将不会正确地址执行。

## 8.5. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 创建一个通过共享内存通讯的客户端/服务器应用程序。 文件名称应该作为命令行参数传递给客户端应用程序。 这个文件存储在服务器端应用程序启动的目录下，这个文件应经由共享内存发送给服务器端应用程序。

# 第 9 章 文件系统

## 目录







## 9.1. 概述

库 Boost.Filesystem 简化了处理文件和目录的工作。 它提供了一个名为 `boost::filesystem::path` 的类，可以对路径进行处理。 另外，还有多个函数用于创建目录或验证某个给定文件的有效性。

## 9.2. 路径

`boost::filesystem::path` 是 Boost.Filesystem 中的核心类，它表示路径的信息，并提供了处理路径的方法。

实际上， `boost::filesystem::path` 是 boost::filesystem::basic_path<std::string> 的一个 `typedef` 。 此外还有一个 `boost::filesystem::wpath` 是 boost::filesystem::basic_path<std::wstring> 的 `typedef` 。

所有定义均位于 boost::filesystem 名字空间，定义于 `boost/filesystem.hpp` 中。

可以通过传入一个字符串至 `boost::filesystem::path` 类来构建一个路径。

```
#include <boost/filesystem.hpp>

int main()
{
  boost::filesystem::path p1("C:\\");
  boost::filesystem::path p2("C:\\Windows");
  boost::filesystem::path p3("C:\\Program Files");
}
```

- 下载源代码

没有一个 `boost::filesystem::path` 的构造函数会实际验证所提供路径的有效性，或检查给定的文件或目录是否存在。 因此， `boost::filesystem::path` 甚至可以用无意义的路径来初始化。

```
#include <boost/filesystem.hpp>

int main()
{
  boost::filesystem::path p1("...");
  boost::filesystem::path p2("\\");
  boost::filesystem::path p3("@:");
}
```

- 下载源代码

以上程序可以执行的原因是，路径其实只是字符串而已。 `boost::filesystem::path` 只是处理字符串罢了；文件系统没有被访问到。

`boost::filesystem::path` 特别提供了一些方法来以字符串方式获取一个路径。 有趣的是，有三种不同的方法。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\Windows\\System");
  std::cout << p.string() << std::endl;
  std::cout << p.file_string() << std::endl;
  std::cout << p.directory_string() << std::endl;
}
```

- 下载源代码

`string()` 方法返回一个所谓的可移植路径。 换句话说，就是 Boost.Filesystem 用它自己预定义的规则来正规化给定的字符串。 在以上例子中， `string()` 返回 `C:/Windows/System` 。 如你所见，Boost.Filesystem 内部使用斜杠符 `/` 作为文件名与目录名的分隔符。

可移植路径的目的是在不同的平台，如 Windows 或 Linux 之间，唯一地标识文件和目录。 因此就不再需要使用预处理器宏来根据底层的操作系统进行路径的编码。 构建可移植路径的规则大多符合POSIX标准，在 Boost.Filesystem 参考手册 给出。

请注意， `boost::filesystem::path` 的构造函数同时支持可移植路径和平台相关路径。 在上面例子中所使用的路径 "C:\Windows\System" 就不是可移植路径，而是 Windows 专用的。 它可以被 Boost.Filesystem 正确识别，但仅当该程序是在 Windows 操作系统下运行的时候！ 当程序运行于一个 POSIX 兼容的操作系统，如

Linux 时， `string()` 将返回 `C:\Windows\System` 。 因为在 Linux 中，反斜杠符 `\` 并不被用作分隔符，无论是可移植格式或原生格式，Boost.Filesystem 都不会认为它是文件和目录的分隔符。

很多时候，都不能避免使用平台相关路径作为字符串。 一个例子就是，使用操作系统函数时必须要用平台相关的编码。 方法 `file_string()` 和 `directory_string()` 正是为此目的而提供的。

在上例中，这两个方法都会返回 `C:\Windows\System` - 与底层操作系统无关。 在 Windows 上这个字符串是有效路径，而在一个 Linux 系统上则既不是可移植路径也不是平台相关路径，会象前面所说那样被解析。

以下例子使用一个可移植路径来初始化 `boost::filesystem::path` 。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("/");
  std::cout << p.string() << std::endl;
  std::cout << p.file_string() << std::endl;
  std::cout << p.directory_string() << std::endl;
}
```

- 下载源代码

由于 `string()` 返回的是一个可移植路径，所以它与用于初始化 `boost::filesystem::path` 的字符串相同： `/` 。 但是 `file_string()` 和 `directory_string()` 方法则会因底层平台而返回不同的结果。 在 Windows 中，它们都返回 `\` ，而在 Linux 中则都返回 `/` 。

你可能会奇怪为什么会有两个不同的方法用来返回平台相关路径。 到目前为止，在所看到的例子中， `file_string()` 和 `directory_string()` 都是返回相同的值。 但是，有些操作系统可能会返回不同的结果。 因为 Boost.Filesystem 的目标是支持尽可能多的操作系统，所以它提供了两个方法来适应这种情况。 即使你可能更为熟悉 Windows 或 POSIX 系统如 Linux，但还是建议使用 `file_string()` 来取出文件的路径信息，且使用 `directory_string()` 取出目录的路径信息。这无疑会增加代码的可移植性。

`boost::filesystem::path` 提供了几个方法来访问一个路径中的特定组件。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\Windows\\System");
  std::cout << p.root_name() << std::endl;
  std::cout << p.root_directory() << std::endl;
  std::cout << p.root_path() << std::endl;
  std::cout << p.relative_path() << std::endl;
  std::cout << p.parent_path() << std::endl;
  std::cout << p.filename() << std::endl;
}
```

- 下载源代码

如果在是一个 Windows 操作系统上执行，则字符串 "C:\Windows\System" 被解释为一个平台相关的路径信息。 因此， `root_name()` 返回 `C:` ， `root_directory()` 返回 `/` ， `root_path()` 返回 `C:/` ， `relative_path()` 返回 `Windows/System` ， `parent_path()` 返回 `C:/Windows` ，而 `filename()` 返回 `System` 。

如你所见，没有平台相关的路径信息被返回。 没有一个返回值包含反斜杠 `\` ，只有斜杠 `/` 。 如果需要平台相关信息，则要使用 `file_string()` 或 `directory_string()` 。 为了使用这些路径中的单独组件，必须创建一个类型为 `boost::filesystem::path` 的新对象并相应的进行初始化。

如果以上程序在 Linux 操作系统中执行，则返回值有所不同。 多数方法会返回一个空字符串，除了 `relative_path()` 和 `filename()` 会返回 `C:\Windows\System` 。 字符串 "C:\Windows\System" 在 Linux 中被解释为一个文件名，这个字符串既不是某个路径的可移植编码，也不是一个被 Linux 支持的平台相关编码。 因此，Boost.Filesystem 没有其它选择，只能将整个字符串解释为一个文件名。

Boost.Filesystem 还提供了其它方法来检查一个路径中是否包含某个特定子串。 这些方法是： `has_root_name()` ， `has_root_directory()` ， `has_root_path()` ， `has_relative_path()` ， `has_parent_path()` 和 `has_filename()` 。 各个方法都是返回一个 `bool` 类型的值。

还有两个方法用于将一个文件名拆分为各个组件。 它们应当仅在 `has_filename()` 返回 `true` 时使用。 否则只会返回一个空字符串，因为如果没有文件名就没什么可拆分了。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("photo.jpg");
  std::cout << p.stem() << std::endl;
  std::cout << p.extension() << std::endl;
}
```

- 下载源代码

这个程序分别返回 `photo` 给 `stem()` ，以及 `.jpg` 给 `extension()` 。

除了使用各个方法调用来访问路径的各个组件以外，你还可以对组件本身进行迭代。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\Windows\\System");
  for (boost::filesystem::path::iterator it = p.begin(); it != p.en
    std::cout << *it << std::endl;
}
```

- 下载源代码

如果是在 Windows 上执行，则该程序将相继输出 `C:` , `/` , `Windows` 和 `System` 。 在其它的操作系统如 Linux 上，输出结果则是 `C:\Windows\System` 。

前面的例子示范了不同的方法来访问路径中的各个组件，以下例子则示范了修改路径信息的方法。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\");
  p /= "Windows\\System";
  std::cout << p.string() << std::endl;
}
```

- 下载源代码

通过使用重载的 `operator/=()` 操作符，这个例子将一个路径添加到另一个之上。 在 Windows 中，该程序将输出 `C:\Windows\System` 。 在 Linux 中，输出将会是 `C:\/Windows\System` ，因为斜杠符 `/` 是文件与目录的分隔符。 这也是重载 `operator/=()` 操作符的原因：毕竟，斜杠是这个方法名的一个部分。

除了 `operator/=()` ，Boost.Filesystem 只提供了 `remove_filename()` 和 `replace_extension()` 方法来修改路径信息。

## 9.3. 文件与目录

`boost::filesystem::path` 的各个方法内部其实只是对字符串进行处理。 它们可以用来访问一个路径的各个组件、相互添加路径等等。

为了处理硬盘上的物理文件和目录，提供了几个独立的函数。 这些函数需要一个或多个 `boost::filesystem::path` 类型的参数，并且在其内部会调用操作系统功能来处理这些文件或目录。

在介绍各个函数之前，很重要的一点是要弄明白出现错误时会发生什么。 所有要在内部访问操作系统功能的函数都有可能失败。 在失败的情况下，将抛出一个类型为 `boost::filesystem::filesystem_error` 的异常。 这个类是派生自 `boost::system::system_error` 的，因此适用于 Boost.System 框架。

除了继承自父类 `boost::system::system_error` 的 `what()` 和 `code()` 方法以外，还有另外两个方法： `path1()` 和 `path2()` 。 它们均返回一个类型为 `boost::filesystem::path` 的对象，因此在发生错误时可以很容易地确定路径信息 - 即使是对那些需要两个 `boost::filesystem::path` 参数的函数。

多数函数存在两个变体：在失败时，一个会抛出类型为 `boost::filesystem::filesystem_error` 的异常，而另一个则返回类型为 `boost::system::error_code` 的对象。 对于后者，需要对返回值进行明确的检查以确定是否出错。

以下例子介绍了一个函数，它可以查询一个文件或目录的状态。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\");
  try
  {
    boost::filesystem::file_status s = boost::filesystem::status(p
    std::cout << boost::filesystem::is_directory(s) << std::endl;
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

`boost::filesystem::status()` 返回一个 `boost::filesystem::file_status` 类型的对象，该对象可以被传递给其它辅助函数来评估。 例如，如果查询的是一个目录的状态，则 `boost::filesystem::is_directory()` 将返回 `true` 。 除了 `boost::filesystem::is_directory()` ，还有其它函数，如 `boost::filesystem::is_regular_file()` ， `boost::filesystem::is_symlink()` 和 `boost::filesystem::exists()` ，它们都会返回一个 `bool` 类型的值。

除了 `boost::filesystem::status()` ，另一个名为 `boost::filesystem::symlink_status()` 的函数可用于查询一个符号链接的状态。 在此情况下，实际上查询的是符号链接所指向的文件的状态。在 Windows 中，符号链接以 `lnk` 文件扩展名识别。

另有一组函数可用于查询文件和目录的属性。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\Windows\\win.ini");
  try
  {
    std::cout << boost::filesystem::file_size(p) << std::endl;
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

函数 `boost::filesystem::file_size()` 以字节数返回一个文件的大小。

```
#include <boost/filesystem.hpp>
#include <iostream>
#include <ctime>

int main()
{
  boost::filesystem::path p("C:\\Windows\\win.ini");
  try
  {
    std::time_t t = boost::filesystem::last_write_time(p);
    std::cout << std::ctime(&t) << std::endl;
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

要获得一个文件最后被修改的时间，可使用 `boost::filesystem::last_write_time()` 。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\");
  try
  {
    boost::filesystem::space_info s = boost::filesystem::space(p);
    std::cout << s.capacity << std::endl;
    std::cout << s.free << std::endl;
    std::cout << s.available << std::endl;
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

`boost::filesystem::space()` 用于取回磁盘的总空间和剩余空间。 它返回一个 `boost::filesystem::space_info` 类型的对象，其中定义了三个公有属性： `capacity` ， `free` 和 `available` 。 这三个属性的类型均为 `boost::uintmax_t` ，该类型定义于 Boost.Integer 库，通常是 `unsigned long long` 的 typedef。 磁盘空间是以字节数来计算的。

目前所看到的函数都不会触及文件和目录本身，不过有另外几个函数可以用于创建、改名或删除文件和目录。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("C:\\Test");
  try
  {
    if (boost::filesystem::create_directory(p))
    {
      boost::filesystem::rename(p, "C:\\Test2");
      boost::filesystem::remove("C:\\Test2");
    }
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

以上例子应该是自解释的。仔细察看，可以看到传递给各个函数的不一定是 `boost::filesystem::path` 类型的对象，也可以是一个简单的字符串。这是可以的，因为 `boost::filesystem::path` 提供了一个非显式的构造函数，可以从简单的字符串转换为 `boost::filesystem::path` 类型的对象。这实际上简化了 Boost.Filesystem 的使用，因为可以无须显式创建一个对象。

还有其它的函数，如 `create_symlink()` 用于创建符号链接，以及 `copy_file()` 用于复制文件或目录。

以下例子中介绍了一个函数，基于一个文件名或一小节路径来创建一个绝对路径。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::filesystem::complete("photo.jpg") << std::e
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

输出哪个路径是由该程序运行时所处的路径决定的。 例如，如果该例子从 `C:\` 运行，输出将是 `C:/photo.jpg` 。

请再次留意斜杠符 `/` ! 如果想得到一个平台相关的路径，则需要初始化一个 `boost::filesystem::path` 类型的对象，且必须调用 `file_string()` 。

要取出一个相对于其它目录的绝对路径，可将第二个参数传递给 `boost::filesystem::complete()` 。

```
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::filesystem::complete("photo.jpg", "D:\\") <
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

现在，该程序显示的是 `D:/photo.jpg` 。

最后，还有一个辅助函数用于取出当前工作目录，如下例所示。

```
#include <windows.h>
#include <boost/filesystem.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::filesystem::current_path() << std::endl;
    SetCurrentDirectory("C:\\");
    std::cout << boost::filesystem::current_path() << std::endl;
  }
  catch (boost::filesystem::filesystem_error &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

以上程序只能在 Windows 中执行，这是 `SetCurrentDirectory()` 函数的原因。 这个函数更换了当前工作目录，因此对 `boost::filesystem::current_path()` 的两次调用将返回不同的结果。

函数 `boost::filesystem::initial_path()` 用于返回应用程序开始执行时所处的目录。 但是，这个函数取决于操作系统的支持，因此如果需要可移植性，建议不要使用。 在这种情况下，Boost.Filesystem 文档中建议的方法是，可以在程序开始时保存 `boost::filesystem::current_path()` 的返回值，以备后用。

## 9.4. 文件流

C++ 标准在 `fstream` 头文件中定义了几个文件流。 这些流不能接受 `boost::filesystem::path` 类型的参数。 由于 Boost.Filesystem 库很有可能被包含在 C++ 标准的 Technical Report 2 中，所以这些文件流将通过相应的构造函数来进行扩展。 为了当前可以让文件流与类型为 `boost::filesystem::path` 的路径信息一起工作，可以使用头文件 `boost/filesystem/fstream.hpp` 。 它提供了对文件流所需的扩展，这些都是基于 Technical Report 2 即将加入 C++ 标准中的。

```
#include <boost/filesystem/fstream.hpp>
#include <iostream>

int main()
{
  boost::filesystem::path p("test.txt");
  boost::filesystem::ofstream ofs(p);
  ofs << "Hello, world!" << std::endl;
}
```

- 下载源代码

不仅是构造函数，还有 `open()` 方法也需要重载，以接受类型为 `boost::filesystem::path` 的参数。

## 9.5. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 创建一个程序，该程序为位于应用程序当前工作目录的上一层目录中的一个名为 `data.txt` 的文件创建一个绝对路径。 例如，如果该程序从 `C:\Program Files\Test` 执行，则应显示 `C:\Program Files\data.txt` 。

# 第 10 章 日期与时间

目录






## 10.1. 概述

库 Boost.DateTime 可用于处理时间数据，如历法日期和时间。 另外，Boost.DateTime 还提供了扩展来处理时区的问题，且支持历法日期和时间的格式化输入与输出。 本章将覆盖 Boost.DateTime 的各个部分。

## 10.2. 历法日期

Boost.DateTime 只支持基于格里历的历法日期，这通常不成问题，因为这是最广泛使用的历法。 如果你与其它国家的某人有个会议，时间在2010年1月5日，你可以期望无需与对方确认这个日期是否基于格里历。

格里历是教皇 Gregory XIII 在1582年颁发的。 严格来说，Boost.DateTime 支持由1400年至9999年的历法日期，这意味着它支持1582年以前的日期。 因此，Boost.DateTime 可用于任一历法日期，只要该日期在转换为格里历后是在1400年之后。 如果需要更早的年份，就必须使用其它库来代替。

用于处理历法日期的类和函数位于名字空间 `boost::gregorian` 中，定义于 `boost/date_time/gregorian/gregorian.hpp` 。 要创建一个日期，请使用 `boost::gregorian::date` 类。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d(2010, 1, 30);
  std::cout << d.year() << std::endl;
  std::cout << d.month() << std::endl;
  std::cout << d.day() << std::endl;
  std::cout << d.day_of_week() << std::endl;
  std::cout << d.end_of_month() << std::endl;
}
```

- 下载源代码

`boost::gregorian::date` 提供了多个构造函数来进行日期的创建。 最基本的构造函数接受一个年份、一个月份和一个日期作为参数。 如果给定的是一个无效值，则将分别抛出 `boost::gregorian::bad_year` ， `boost::gregorian::bad_month` 或 `boost::gregorian::bad_day_of_month` 类型的异常，这些异常均派生自 `std::out_of_range` 。

正如在这个例子中所示的， 有多个方法用于访问一个日期。 象 `year()` ， `month()` 和 `day()` 这些方法访问用于初始化的初始值，象 `day_of_week()` 和 `end_of_month()` 这些方法则访问计算得到的值。

而 `boost::gregorian::date` 的构造函数则接受年份、月份和日期的值来设定一个日期，调用 `month()` 方法实际上会显示 `Jan` ，而调用 `day_of_week()` 则显示 `Sat` 。 它们不是普通的数字值，而分别是 `boost::gregorian::date::month_type` 和 `boost::gregorian::date::day_of_week_type` 类型的值。 不过，Boost.DateTime 为格式化的输入输出提供了全面的支持，可以将以上输出从 `Jan` 调整为 `1` 。

请留意， `boost::gregorian::date` 的缺省构造函数会创建一个无效的日期。 这样的无效日期也可以通过将 `boost::date_time::not_a_date_time` 作为单一参数传递给构造函数来显式地创建。

除了直接调用构造函数，也可以通过自由函数或其它对象的方法来创建一个 `boost::gregorian::date` 类型的对象。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d = boost::gregorian::day_clock::universal
  std::cout << d.year() << std::endl;
  std::cout << d.month() << std::endl;
  std::cout << d.day() << std::endl;

  d = boost::gregorian::date_from_iso_string("20100131");
  std::cout << d.year() << std::endl;
  std::cout << d.month() << std::endl;
  std::cout << d.day() << std::endl;
}
```

- 下载源代码

这个例子使用了 `boost::gregorian::day_clock` 类，它是一个返回当前日期的时钟类。方法 `universal_day()` 返回一个与时区及夏时制无关的 UTC 日期。UTC 是世界时(universal time)的国际缩写。 `boost::gregorian::day_clock` 还提供了另一个方法 `local_day()` ，它接受本地设置。要取出本地时区的当前日期，必须使用 `local_day()` 。

名字空间 `boost::gregorian` 中包含了许多其它自由函数，将保存在字符串中的日期转换为 `boost::gregorian::date` 类型的对象。这个例子实际上是通过 `boost::gregorian::date_from_iso_string()` 函数对一个以 ISO 8601 格式给出的日期进行转换。还有其它相类似的函数，如 `boost::gregorian::from_simple_string()` 和 `boost::gregorian::from_us_string()` 。

`boost::gregorian::date` 表示的是一个特定的时间点，而 `boost::gregorian::date_duration` 则表示了一段时间。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d1(2008, 1, 31);
  boost::gregorian::date d2(2008, 8, 31);
  boost::gregorian::date_duration dd = d2 - d1;
  std::cout << dd.days() << std::endl;
}
```

- 下载源代码

由于 `boost::gregorian::date` 重载了 `operator-()` 操作符，所以两个时间点可以如上所示那样相减。 返回值的类型为 `boost::gregorian::date_duration` ，表示了两个日期之间的时间长度。

`boost::gregorian::date_duration` 所提供的最重要的方法是 `days()` ，它返回一段时间内所包含的天数。

我们也可以通过传递一个天数作为构造函数的唯一参数，来显式创建 `boost::gregorian::date_duration` 类型的对象。 要创建涉及星期数、月份数或年数的时间段，可以相应使用 `boost::gregorian::weeks` ， `boost::gregorian::months` 和 `boost::gregorian::years` 。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date_duration dd(4);
  std::cout << dd.days() << std::endl;
  boost::gregorian::weeks ws(4);
  std::cout << ws.days() << std::endl;
  boost::gregorian::months ms(4);
  std::cout << ms.number_of_months() << std::endl;
  boost::gregorian::years ys(4);
  std::cout << ys.number_of_years() << std::endl;
}
```

- 下载源代码

`boost::gregorian::months` 和 `boost::gregorian::years` 都无法确定其天数，因为某月或某年所含天数是可长的。 不过，这些类的用法还是可以从以下例子中看出。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d(2009, 1, 31);
  boost::gregorian::months ms(1);
  boost::gregorian::date d2 = d + ms;
  std::cout << d2 << std::endl;
  boost::gregorian::date d3 = d2 - ms;
  std::cout << d3 << std::endl;
}
```

- 下载源代码

该程序将一个月加到给定的日期 January 31, 2009 上，得到 `d2`，其为 February 28, 2009。 接着，再减回一个月得到 `d3`，又重新变回 January 31, 2009。 如上所示，时间点和时间长度可用于计算。 不过，需要考虑具体的情况。 例如，从某月的最后一天开始计算，`boost::gregorian::months` 总是会到达另一个月的最后一天，如果反复前后跳，就可能得到令人惊讶的结果。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d(2009, 1, 30);
  boost::gregorian::months ms(1);
  boost::gregorian::date d2 = d + ms;
  std::cout << d2 << std::endl;
  boost::gregorian::date d3 = d2 - ms;
  std::cout << d3 << std::endl;
}
```

- 下载源代码

这个例子与前一个例子的不同之处在于，初始的日期是 January 30, 2009。 虽然这不是 January 的最后一天，但是向前跳一个月后得到的 `d2` 还是 February 28, 2009，因为没有 February 30 这一天。 不过，当我们再往回跳一个月，这次得到的 `d3` 就变成 January 31, 2009! 因为 February 28, 2009 是当月的最后一天，往回跳实际上是返回到 January 的最后一天。

如果你觉得这种行为过于混乱，可以通过取消 `BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES` 宏的定义来改变这种行为。 取消该宏后，`boost::gregorian::weeks`，`boost::gregorian::months` 和 `boost::gregorian::years` 类都不再可用。 唯一剩下的类是 `boost::gregorian::date_duration`，只能指定前向或后向的跳过的天数，这样就不会再有意外的结果了。

`boost::gregorian::date_duration` 表示的是时间长度，而 `boost::gregorian::date_period` 则提供了对两个日期之间区间的支持。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d1(2009, 1, 30);
  boost::gregorian::date d2(2009, 10, 31);
  boost::gregorian::date_period dp(d1, d2);
  boost::gregorian::date_duration dd = dp.length();
  std::cout << dd.days() << std::endl;
}
```

- 下载源代码

两个类型为 `boost::gregorian::date` 的参数指定了开始和结束的日期，它们被传递给 `boost::gregorian::date_period` 的构造函数。 此外，也可以指定一个开始日期和一个类型为 `boost::gregorian::date_duration` 的时间长度。 请注意，结束日期的前一天才是这个时间区间的最后一天，这对于理解以下例子的输出非常重要。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d1(2009, 1, 30);
  boost::gregorian::date d2(2009, 10, 31);
  boost::gregorian::date_period dp(d1, d2);
  std::cout << dp.contains(d1) << std::endl;
  std::cout << dp.contains(d2) << std::endl;
}
```

- 下载源代码

这个程序用 `contains()` 方法来检查某个给定的日期是否包含在时间区间内。 虽然 `d1` 和 `d2` 都是被传递给 `boost::gregorian::date_period` 的构造函数的，但是 `contains()` 仅对第一个返回 `true` 。 因为结束日期不是区间的一部分，所以以 `d2` 调用 `contains()` 会返回 `false` 。

`boost::gregorian::date_period` 还提供了其它方法，如移动一个区间，或计算两个重叠区间的交集。

除了日期类、时间长度类和时间区间类，Boost.DateTime 还提供了迭代器和其它有用的自由函数，如下例所示。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d(2009, 1, 5);
  boost::gregorian::day_iterator it(d);
  std::cout << *++it << std::endl;
  std::cout << boost::date_time::next_weekday(*it, boost::gregorian
}
```

- 下载源代码

为了从一个指定日期向前或向后一天一天地跳，可以使用迭代器 `boost::gregorian::day_iterator` 。 还有 `boost::gregorian::week_iterator` , `boost::gregorian::month_iterator` 和 `boost::gregorian::year_iterator` 分别提供了按周、按月或是按年跳的迭代器。

这个例子还使用了 `boost::date_time::next_weekday()` ，它基于一个给定的日期返回下一个星期几的日期。 以下程序将显示 `2009-Jan-09` ，因为它是 January 6, 2009 之的第一个Friday。

## 10.3. 位置无关的时间

`boost::gregorian::date` 用于创建日期, `boost::posix_time::ptime` 则用于定义一个位置无关的时间。 `boost::posix_time::ptime` 会存取 `boost::gregorian::date` 且额外保存一个时间。

为了使用 `boost::posix_time::ptime` ，必须包含头文件 `boost/date_time/posix_time/posix_time.hpp` 。

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::posix_time::ptime pt(boost::gregorian::date(2009, 1, 5), b
  boost::gregorian::date d = pt.date();
  std::cout << d << std::endl;
  boost::posix_time::time_duration td = pt.time_of_day();
  std::cout << td << std::endl;
}
```

- 下载源代码

要初始化一个 `boost::posix_time::ptime` 类型的对象，要把一个类型为 `boost::gregorian::date` 的日期和一个类型为 `boost::posix_time::time_duration` 的时间长度作为第一和第二参数传递给构造函数。 传给 `boost::posix_time::time_duration` 构造函数的三个参数决定了时间点。 以上程序指定的时间点是 January 5, 2009 的 12 PM。

要查询日期和时间，可以使用 `date()` 和 `time_of_day()` 方法。

象 `boost::gregorian::date` 的缺省构造函数会创建一个无效日期一样，如果使用缺省构造函数， `boost::posix_time::ptime` 类型的对象也是无效的。 也可以通过传递一个 `boost::date_time::not_a_date_time` 给构造函数来显式创建一个无效时间。

和使用自由函数或其它对象的方法来创建 `boost::gregorian::date` 类型的历法日期一样，Boost.DateTime 也提供了相应的自由函数和对象来创建时间。

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::posix_time::ptime pt = boost::posix_time::second_clock::ur
  std::cout << pt.date() << std::endl;
  std::cout << pt.time_of_day() << std::endl;

  pt = boost::posix_time::from_iso_string("20090105T120000");
  std::cout << pt.date() << std::endl;
  std::cout << pt.time_of_day() << std::endl;
}
```

- 下载源代码

类 `boost::posix_time::second_clock` 表示一个返回当前时间的时钟。 `universal_time()` 方法返回 UTC 时间，如上例所示。 如果需要本地时间，则必须使用 `local_time()` 。

Boost.DateTime 还提供了一个名为 `boost::posix_time::microsec_clock` 的类，它返回包含微秒在内的当前时间，供需要更高精度时使用。

要将一个保存在字符串中的时间点转换为类型为 `boost::posix_time::ptime` 的对象，可以用 `boost::posix_time::from_iso_string()` 这样的自由函数，它要求传入的时间点以 ISO 8601 格式提供。

除了 `boost::posix_time::ptime` , Boost.DateTime 也提供了 `boost::posix_time::time_duration` 类，用于指定一个时间长度。 这个类前面已经提到过，因为 `boost::posix_time::ptime` 的构造函数实际上需要一个 `boost::posix_time::time_duration` 类型的对象作为其第二个参数。 当然， `boost::posix_time::time_duration` 也可以单独使用。

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

int main()
{
  boost::posix_time::time_duration td(16, 30, 0);
  std::cout << td.hours() << std::endl;
  std::cout << td.minutes() << std::endl;
  std::cout << td.seconds() << std::endl;
  std::cout << td.total_seconds() << std::endl;
}
```

- 下载源代码

`hours()` , `minutes()` 和 `seconds()` 均返回传给构造函数的各个值，而象 `total_seconds()` 这样的方法则返回总的秒数，以简单的方式为你提供额外的信息。

可以传递任意值给 `boost::posix_time::time_duration` ，因为没有象24小时这样的上限存在。

和历法日期一样，时间点与时间长度也可以执行运算。

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

int main()
{
  boost::posix_time::ptime pt1(boost::gregorian::date(2009, 1, 05),
  boost::posix_time::ptime pt2(boost::gregorian::date(2009, 1, 05),
  boost::posix_time::time_duration td = pt2 - pt1;
  std::cout << td.hours() << std::endl;
  std::cout << td.minutes() << std::endl;
  std::cout << td.seconds() << std::endl;
}
```

- 下载源代码

如果两个 `boost::posix_time::ptime` 类型的时间点相减，结果将是一个 `boost::posix_time::time_duration` 类型的对象，给出两个时间点之间的时间长度。

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

int main()
{
  boost::posix_time::ptime pt1(boost::gregorian::date(2009, 1, 05),
  boost::posix_time::time_duration td(6, 30, 0);
  boost::posix_time::ptime pt2 = pt1 + td;
  std::cout << pt2.time_of_day() << std::endl;
}
```

- 下载源代码

正如这个例子所示，时间长度可以被增加至一个时间点上，以得到一个新的时间点。 以上程序将打印 `18:30:00` 到标准输出流。

你可能已经留意到，Boost.DateTime 对于历法日期和时间使用了相同的概念。 就象有时间类和时间长度类一样，也有一个时间区间的类。 对于历法日期，这个类是 `boost::gregorian::date_period` ; 对于时间则是 `boost::posix_time::time_period` 。 这两个类均要求传入两个参数给构造函数： `boost::gregorian::date_period` 要求两个历法日期，而 `boost::posix_time::time_period` 则要求两个时间。

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

int main()
{
  boost::posix_time::ptime pt1(boost::gregorian::date(2009, 1, 05),
  boost::posix_time::ptime pt2(boost::gregorian::date(2009, 1, 05),
  boost::posix_time::time_period tp(pt1, pt2);
  std::cout << tp.contains(pt1) << std::endl;
  std::cout << tp.contains(pt2) << std::endl;
}
```

- 下载源代码

大致上说， `boost::posix_time::time_period` 非常象 `boost::gregorian::date_period` 。 它提供了一个名为 `contains()` 的方法，对于位于该时间区间内的每一个时间点，它返回 `true` 。 由于传给 `boost::posix_time::time_period` 的构造函数的结束时间不是时间区间的一部分，所以上例中第二个 `contains()` 调用将返回 `false` 。

`boost::posix_time::time_period` 还提供了其它方法，如 `intersection()` 和 `merge()` 分别用于计算两个重叠时间区间的交集，以及合并两个相交区间。

最后，迭代器 `boost::posix_time::time_iterator` 用于对时间点进行迭代。

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

int main()
{
  boost::posix_time::ptime pt(boost::gregorian::date(2009, 1, 05),
  boost::posix_time::time_iterator it(pt, boost::posix_time::time_(
  std::cout << *++it << std::endl;
  std::cout << *++it << std::endl;
}
```

- 下载源代码

以上程序使用了迭代器 `it` 从时间点 `pt` 开始向前跳6.5个小时 。 由于迭代器被递增两次，所以相应的输出分别为 `2009-Jan-05 18:30:00` 和 `2009-Jan-06 01:00:00` 。

## 10.4. 位置相关的时间

和前一节所介绍的位置无关时间不一样，位置相关时间是要考虑时区的。 为此，Boost.DateTime 提供了 `boost::local_time::local_date_time` 类，它定义于 `boost/date_time/local_time/local_time.hpp` ，并使用 `boost::local_time::posix_time_zone` 来保存时区信息。

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

int main()
{
  boost::local_time::time_zone_ptr tz(new boost::local_time::posix_
  boost::posix_time::ptime pt(boost::gregorian::date(2009, 1, 5), b
  boost::local_time::local_date_time dt(pt, tz);
  std::cout << dt.utc_time() << std::endl;
  std::cout << dt << std::endl;
  std::cout << dt.local_time() << std::endl;
  std::cout << dt.zone_name() << std::endl;
}
```

- 下载源代码

`boost::local_time::local_date_time` 的构造函数要求一个 `boost::posix_time::ptime` 类型的对象作为其第一个参数，以及一个 `boost::local_time::time_zone_ptr` 类型的对象作为第二个参数。 后者只不过是 boost::shared_ptr<boost::local_time::posix_time_zone> 的类型定义。 换句话说，并不是传递一个 `boost::local_time::posix_time_zone` 对象，而是传递一个指向该对象的智能指针。 这样，多个 `boost::local_time::local_date_time` 类型的对象就可以共享时区信息。 只有当最后一个对象被销毁时，相应的表示时区的对象才会被自动释放。

要创建一个 `boost::local_time::posix_time_zone` 类型的对象，就要将一个描述该时区的字符串作为单一参数传递给构造函数。 以上例子指定了欧洲中部时区：CET 是欧洲中部时间(Central European Time)的缩写。 由于 CET 比 UTC 早一个小时，所以时差以 +1 表示。 Boost.DateTime 并不能解释时区的缩写，也就不知道 CET 的意思。 所以，必须以小时数给出时差；传入 +0 表示没有时差。

在执行时，该程序将打印 `2009-Jan-05 12:00:00` ， `2009-Jan-05 13:00:00 CET` ， `2009-Jan-05 13:00:00` 和 `CET` 到标准输出流。 用以初始化 `boost::posix_time::ptime` 和

`boost::local_time::local_date_time` 类型的值缺省总是与 UTC 时区相关的。 只有当一个 `boost::local_time::local_date_time` 类型的对象被写出至标准输出流时，或者调用 `local_time()` 方法时，才使用时差来计算本地时间。

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

int main()
{
  boost::local_time::time_zone_ptr tz(new boost::local_time::posix_
  boost::posix_time::ptime pt(boost::gregorian::date(2009, 1, 5), b
  boost::local_time::local_date_time dt(pt, tz);
  std::cout << dt.local_time() << std::endl;
  boost::local_time::time_zone_ptr tz2(new boost::local_time::posix
  std::cout << dt.local_time_in(tz2).local_time() << std::endl;
}
```

- 下载源代码

通过使用 `local_time()` 方法，时区的偏差才被考虑进来。 为了计算 CET 时间，需要往保存在 `dt` 中的 UTC 时间 12 PM 上加一个小时，因为 CET 比 UTC 早一个小时。 `local_time()` 会相应地输出 `2009-Jan-05 13:00:00` 到标准输出流。

相比之下， `local_time_in()` 方法是在所传入参数的时区内解释保存在 `dt` 中的时间。 这意味着 12 PM UTC 相当于 2 PM EET，即东部欧洲时间，它比 UTC 早两个小时。

最后，Boost.DateTime 通过 `boost::local_time::local_time_period` 类提供了位置相关的时间区间。

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

int main()
{
  boost::local_time::time_zone_ptr tz(new boost::local_time::posix_
  boost::posix_time::ptime pt1(boost::gregorian::date(2009, 1, 5),
  boost::local_time::local_date_time dt1(pt1, tz);
  boost::posix_time::ptime pt2(boost::gregorian::date(2009, 1, 5),
  boost::local_time::local_date_time dt2(pt2, tz);
  boost::local_time::local_time_period tp(dt1, dt2);
  std::cout << tp.contains(dt1) << std::endl;
  std::cout << tp.contains(dt2) << std::endl;
}
```

- 下载源代码

`boost::local_time::local_time_period` 的构造函数要求两个类型为 `boost::local_time::local_date_time` 的参数。和其它类型的时间区间一样，第二个参数所表示的结束时间并不包含在区间之内。通过如 `contains()`，`intersection()`，`merge()` 以及其它的方法，时间区间可以与其它 `boost::local_time::local_time_period` 相互操作。

## 10.5. 格式化输入输出

本章中的所有例子在执行后都提供形如 `2009-Jan-07` 这样的输出结果。有的人可能更喜欢用其它格式来显示结果。Boost.DateTime 允许 `boost::date_time::date_facet` 和 `boost::date_time::time_facet` 类来格式化历法日期和时间。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>
#include <locale>

int main()
{
  boost::gregorian::date d(2009, 1, 7);
  boost::gregorian::date_facet *df = new boost::gregorian::date_fac
  std::cout.imbue(std::locale(std::cout.getloc(), df));
  std::cout << d << std::endl;
}
```

- 下载源代码

Boost.DateTime 使用了 locales 的概念，它来自于 C++ 标准，在 第 5 章 字符串处理 中有概括的介绍。要格式化一个历法日期，必须创建一个 `boost::date_time::date_facet` 类型的对象并安装在一个 locale 内。一个描述新格式的字符串被传递给 `boost::date_time::date_facet` 的构造函数。上面的例子传递的是 `%A, %d %B %Y`，指定格式为：星期几后跟日月年全名：`Wednesday, 07 January 2009`。

Boost.DateTime 提供了多个格式化标志，标志由一个百分号后跟一个字符组成。Boost.DateTime 的文档中对于所支持的所有标志有一个完整的介绍。例如，%A 表示星期几的全名。

如果应用程序的基本用户是位于德国或德语国家，最好可以用德语而不是英语来显示星期几和月份。

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>
#include <locale>
#include <string>
#include <vector>

int main()
{
  std::locale::global(std::locale("German"));
  std::string months[12] = { "Januar", "Februar", "März", "April",
  std::string weekdays[7] = { "Sonntag", "Montag", "Dienstag", "Mit
  boost::gregorian::date d(2009, 1, 7);
  boost::gregorian::date_facet *df = new boost::gregorian::date_fac
  df->long_month_names(std::vector<std::string>(months, months + 12
  df->long_weekday_names(std::vector<std::string>(weekdays, weekday
  std::cout.imbue(std::locale(std::cout.getloc(), df));
  std::cout << d << std::endl;
}
```

- 下载源代码

星期几和月份的名字可以通过分别传入两个数组给 `boost::date_time::date_facet` 类的 `long_month_names()` 和 `long_weekday_names()` 方法来修改，这两个数组分别包含了相应的名字。 以上例子现在将打印 `Mittwoch, 07\. Januar 2009` 到标准输出流。

Boost.DateTime 在格式化输入输出方面是非常灵活的。 除了输出类 `boost::date_time::date_facet` 和 `boost::date_time::time_facet` 以外，类 `boost::date_time::date_input_facet` 和 `boost::date_time::time_input_facet` 可用于格式化输入。 所有这四个类都提供了许多方法，来为 Boost.DateTime 所提供的各种不同对象配置输入和输出的方式。 例如，可以指定 `boost::gregorian::date_period` 类型的时间长度如何输入和输出。 要弄清楚各种格式化输入输出的可能性，请参考 Boost.DateTime 的文档。

## 10.6. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 创建一个程序，打印下一个 Christmas Eve, Christmas Day 及其后一天是星期几到标准输出流。
2. 以天数计算你的年龄。 该程序应该自动根据当前日期来计算。

# 第 11 章 序列化

## 目录





## 11.1. 概述

Boost C++ 的 序列化 库允许将 C++ 应用程序中的对象转换为一个字节序列， 此序列可以被保存，并可在将来恢复对象的时候再次加载。 各种不同的数据格式，包括 XML，只要具有一定规则的数据格式，在序列化后都产生一个字节序列。所有 Boost.Serialization 支持的格式，在某些方面来说都是专有的。 比如 XML 格式不同用来和不是用 C++ Boost.Serialization 库开发的应用程序交换数据。所有以 XML 格式存储的数据适合于从之前存储的数据上恢复同一个 C++ 对象。 XML 格式的唯一优点是序列化的 C++ 对象容易理解，这是很有用的，比如说在调试的时候。

## 11.2. 归档

Boost.Serialization 的主要概念是归档。 归档的文件是相当于序列化的 C++ 对象的一个字节流。 对象可以通过序列化添加到归档文件，相应地也可从归档文件中加载。 为了恢复和之前存储相同的 C++ 对象，需假定数据类型是相同的。

下面看一个简单的例子。

```
#include <boost/archive/text_oarchive.hpp>
#include <iostream>

int main()
{
  boost::archive::text_oarchive oa(std::cout);
  int i = 1;
  oa << i;
}
```

- 下载源代码

Boost.Serialization 提供了多个归档类，如 `boost::archive::text_oarchive` 类，它定义在 `boost/archive/text_oarchive.hpp` 文件中。 `boost::archive::text_oarchive` ，可将对象序列化为文本流。 上面的应用程序将 `22 serialization::archive 5 1` 写出到标准输出流。

可见， `boost::archive::text_oarchive` 类型的对象 `oa` 可以用来像流 (stream) 一样通过 << 来序列化对象。 尽管如此，归档也不能被认为是可以存储任何数据的常规的流。 为了以后恢复数据，必须以相同的顺序使用和先前存储时用的一样的数据类型。 下面的例子序列化和恢复了 `int` 类型的变量。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <fstream>

void save()
{
  std::ofstream file("archiv.txt");
  boost::archive::text_oarchive oa(file);
  int i = 1;
  oa << i;
}

void load()
{
  std::ifstream file("archiv.txt");
  boost::archive::text_iarchive ia(file);
  int i = 0;
  ia >> i;
  std::cout << i << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

当 `boost::archive::text_oarchive` 被用来把数据序列化为文本流， `boost::archive::text_iarchive` 就用来从文本流恢复数据。 为了使用这些类，必须包含 `boost/archive/text_iarchive.hpp` 头文件。

归档的构造函数需要一个输入或者输出流作为参数。 流分别用来序列化或恢复数据。 虽然上面的应用程序使用了一个文件流，其他流，如 stringstream 流也是可以的。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

void save()
{
  boost::archive::text_oarchive oa(ss);
  int i = 1;
  oa << i;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  int i = 0;
  ia >> i;
  std::cout << i << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

这个应用程序也向标准输出流写了 `1` 。 然而，与前面的例子相比, 数据却是用 stringstream 流序列化的。

到目前为止， 原始的数据类型已经被序列化了。 接下来的例子演示如何序列化用户定义类型的对象。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
```

```
    : age_(age)
  {
  }

  int age() const
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & age_;
  }

  int age_;
};

void save()
{
  boost::archive::text_oarchive oa(ss);
  person p(31);
  oa << p;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  person p;
  ia >> p;
  std::cout << p.age() << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

为了序列化用户定义类型的对话， `serialize()` 函数必须定义，它在对象序列化或从字节流中恢复是被调用。 由于 `serialize ()` 函数既用来序列化又用来恢复数据， Boost.Serialization 除了 `<<` 和 `>>` 之外还提供了 `&` 操作符。如果使用这个操作符，就不再需要在 `serialize()` 函数中区分是序列化和恢复了。

`serialize ()` 在对象序列化或恢复时自动调用。它应从来不被明确地调用，所以应生命为私有的。 这样的话， `boost::serialization::access` 类必须被声明为友元，以允许 Boost.Serialization 能够访问到这个函数。

有些情况下需要添加 `serialize()` 函数却不能修改现有的类。 比如，对于来自 C++ 标准库或其他库的类就是这样的。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  int age() const
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  friend void serialize(Archive &ar, person &p, const unsigned int

  int age_;
};

template <typename Archive>
void serialize(Archive &ar, person &p, const unsigned int version)
{
  ar & p.age_;
}

void save()
{
  boost::archive::text_oarchive oa(ss);
  person p(31);
  oa << p;
}
```

```
void load()
{
  boost::archive::text_iarchive ia(ss);
  person p;
  ia >> p;
  std::cout << p.age() << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

为了序列化那些不能被修改的数据类型，要定义一个单独的函数 `serialize()` ，如上面的例子所示。 这个函数需要相应的数据类型的引用作为它的第二个参数。

如果要被序列化的数据类型中含有不能经由公有函数访问的私有属性，事情就变得复杂了。 在这种情况下，该数据列席就需要修改。 在上面应用程序中的 `serialize ()` 函数如果不声明为 `friend` ，就不能访问 `age_` 属性。

不过还好，Boost.Serialization 为许多C++标准库的类提供了 `serialize()` 函数。 为了序列化基于 C++ 标准库的类，需要包含额外的头文件。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age, const std::string &name)
    : age_(age), name_(name)
  {
  }

  int age() const
  {
```

```
    return age_;
  }

  std::string name() const
  {
    return name_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  friend void serialize(Archive &ar, person &p, const unsigned int

  int age_;
  std::string name_;
};

template <typename Archive>
void serialize(Archive &ar, person &p, const unsigned int version)
{
  ar & p.age_;
  ar & p.name_;
}

void save()
{
  boost::archive::text_oarchive oa(ss);
  person p(31, "Boris");
  oa << p;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  person p;
  ia >> p;
  std::cout << p.age() << std::endl;
  std::cout << p.name() << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

这个例子扩展了 `person` 类，增加了 `std::string` 类型的名称变量，为了序列化这个属性property, the header file `boost/serialization/string.hpp` 头文件必须包含，它提供了合适的单独的 `serialize ()` 函数。

正如前面所提到的， Boost.Serialization 为许多 C++ 标准库类定义了 `serialize ()` 函数。 这些都定义在和 C++ 标准库头文件名称相对应的头文件中。 为了序列化 `std::string` 类型的对象，必须包含 `boost/serialization/string.hpp` 头文件。 为了序列化 `std::vector` 类型的对象，必须包含 `boost/serialization/vector.hpp` 头文件。 于是在给定的场合中应该包含哪个头文件就显而易见了。

还有一个 `serialize ()` 函数的参数，到目前为止我们一直忽略没谈到，那就是 `version` 。 如果归档需要向前兼容，以支持给定应用程序的未来版本，那么这个参数就是有意义的。 接下来的例子考虑到 `person` 类的归档需要向前兼容。由于 `person` 的原始版本没有包含任何名称，新版本的 `person` 应该能够处理不带名称的旧的归档。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age, const std::string &name)
    : age_(age), name_(name)
  {
  }

  int age() const
  {
    return age_;
  }

  std::string name() const
  {
    return name_;
  }

private:
  friend class boost::serialization::access;
```

```
  template <typename Archive>
  friend void serialize(Archive &ar, person &p, const unsigned int

  int age_;
  std::string name_;
};

template <typename Archive>
void serialize(Archive &ar, person &p, const unsigned int version)
{
  ar & p.age_;
  if (version > 0)
    ar & p.name_;
}

BOOST_CLASS_VERSION(person, 1)

void save()
{
  boost::archive::text_oarchive oa(ss);
  person p(31, "Boris");
  oa << p;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  person p;
  ia >> p;
  std::cout << p.age() << std::endl;
  std::cout << p.name() << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

`BOOST_CLASS_VERSION` 宏用来指定类的版本号。 上面例子中 `person` 类的版本号设置为1。 如果没有使用 `BOOST_CLASS_VERSION` ， 版本号缺省是0。

版本号存储在归档文件中，因此也就是归档的一部份。 当一个特定类的版本号通过 `BOOST_CLASS_VERSION` 宏，在序列化时给定时， `serialize ()` 函数的 `version` 参数被设为给定值存储在归档中。 如果新版本的 `person` 访问一个包含旧版本序列化对象的归档时， `name_` 由于旧版本不含有这个属性而不能恢复。 通过这种机制，Boost.Serialization 提供了向前兼容归档的支持。

# 11.3. 指针和引用

Boost.Serialization 还能序列化指针和引用。 由于指针存储对象的地址，序列化对象的地址没有什么意义，而是在序列化指针和引用时，对象的引用被自动地序列化。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  int age() const
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & age_;
  }

  int age_;
};

void save()
{
  boost::archive::text_oarchive oa(ss);
  person *p = new person(31);
  oa << p;
  std::cout << std::hex << p << std::endl;
  delete p;
}
```

```
void load()
{
  boost::archive::text_iarchive ia(ss);
  person *p;
  ia >> p;
  std::cout << std::hex << p << std::endl;
  std::cout << p->age() << std::endl;
  delete p;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

上面的应用程序创建了一个新的 `person` 类型的对象，使用 `new` 创建并赋值给指针 `p` 。 是指针 - 而不是 `*p` - 被序列化了。Boost.Serialization 自动地通过 `p` 的引用序列化对象本身而不是对象的地址。

如果归档被恢复， `p` 不必指向相同的地址。 而是创建新对象并将它的地址赋值给 `p` 。 Boost.Serialization 只保证对象和之前序列化的对象相同，而不是地址相同。

由于新式的 C++ 在动态分配内存有关的地方使用 智能指针 (smart pointers) ， Boost.Serialization 对此也提供了相应的支持。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/scoped_ptr.hpp>
#include <boost/scoped_ptr.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  int age() const
```

```
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & age_;
  }

  int age_;
};

void save()
{
  boost::archive::text_oarchive oa(ss);
  boost::scoped_ptr<person> p(new person(31));
  oa << p;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  boost::scoped_ptr<person> p;
  ia >> p;
  std::cout << p->age() << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

例子中使用了智能指针 `boost::scoped_ptr` 来管理动态分配的 `person` 类型的对象。 为了序列化这样的指针，必须包含 `boost/serialization/scoped_ptr.hpp` 头文件。

在使用 `boost::shared_ptr` 类型的智能指针的时候需要序列化，那么必须包含 `boost/serialization/shared_ptr.hpp` 头文件。

下面的应用程序使用引用替代了指针。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
```

```
#include <sstream>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  int age() const
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & age_;
  }

  int age_;
};

void save()
{
  boost::archive::text_oarchive oa(ss);
  person p(31);
  person &pp = p;
  oa << pp;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  person p;
  person &pp = p;
  ia >> pp;
  std::cout << pp.age() << std::endl;
}

int main()
{
  save();
```

```
    load();
  }
```

- 下载源代码

可见，Boost.Serialization 还能没有任何问题地序列化引用。 就像指针一样，引用对象被自动地序列化。

## 11.4. 对象类层次结构的序列化

为了序列化基于类层次结构的对象，子类必须在 `serialize ()` 函数中访问 `boost::serialization::base_object ()` 。 此函数确保继承自基类的属性也能正确地序列化。 下面的例子演示了一个名为 `developer` 类，它继承自类 `person` 。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  int age() const
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & age_;
  }
```

```
  int age_;
};

class developer
  : public person
{
public:
  developer()
  {
  }

  developer(int age, const std::string &language)
    : person(age), language_(language)
  {
  }

  std::string language() const
  {
    return language_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<person>(*this);
    ar & language_;
  }

  std::string language_;
};

void save()
{
  boost::archive::text_oarchive oa(ss);
  developer d(31, "C++");
  oa << d;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  developer d;
  ia >> d;
  std::cout << d.age() << std::endl;
  std::cout << d.language() << std::endl;
}

int main()
{
```

```
    save();
    load();
}
```

- 下载源代码

`person` 和 `developer` 这两个类都包含有一个私有的 `serialize()` 函数，它使得基于其他类的对象能被序列化。 由于 `developer` 类继承自 `person` 类， 所以它的 `serialize()` 函数必须确保继承自 `person` 属性也能被序列化。

继承自基类的属性被序列化是通过在子类的 `serialize()` 函数中用 `boost::serialization::base_object()` 函数访问基类实现的。 在例子中强制要求使用这个函数而不是 `static_cast` 是因为只有 `boost::serialization::base_object()` 才能保证正确地序列化。

动态创建对象的地址可以被赋值给对应的基类类型的指针。 下面的例子演示了 Boost.Serialization 还能够正确地序列化它们。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/export.hpp>
#include <iostream>
#include <sstream>
#include <string>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  virtual int age() const
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
```

```
  {
    ar & age_;
  }

  int age_;
};

class developer
  : public person
{
public:
  developer()
  {
  }

  developer(int age, const std::string &language)
    : person(age), language_(language)
  {
  }

  std::string language() const
  {
    return language_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<person>(*this);
    ar & language_;
  }

  std::string language_;
};

BOOST_CLASS_EXPORT(developer)

void save()
{
  boost::archive::text_oarchive oa(ss);
  person *p = new developer(31, "C++");
  oa << p;
  delete p;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  person *p;
  ia >> p;
```

```
  std::cout << p->age() << std::endl;
  delete p;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

应用程序在 save () 函数创建了 developer 类型的对象并赋值给 person* 类型的指针，接下来通过 << 序列化。

正如在前面章节中提到的，引用对象被自动地序列化。为了让 Boost.Serialization 识别将要序列化的 developer 类型的对象，即使指针是 person* 类型的对象。developer 类需要相应的声明。这是通过这个 BOOST_CLASS_EXPORT 宏实现的，它定义在 boost/serialization/export.hpp 文件中。因为 developer 这个数据类型没有指针形式的定义，所以 Boost.Serialization 没有这个宏就不能正确地序列化 developer 。

如果子类对象需要通过基类的指针序列化，那么 BOOST_CLASS_EXPORT 宏必须要用。

由于静态注册的原因，BOOST_CLASS_EXPORT 的一个缺点是可能有些注册的类最后是不需要序列化的。Boost.Serialization 为这种情况提供一种解决方案。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/export.hpp>
#include <iostream>
#include <sstream>
#include <string>

std::stringstream ss;

class person
{
public:
  person()
  {
  }

  person(int age)
    : age_(age)
  {
  }

  virtual int age() const
```

```
  {
    return age_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & age_;
  }

  int age_;
};

class developer
  : public person
{
public:
  developer()
  {
  }

  developer(int age, const std::string &language)
    : person(age), language_(language)
  {
  }

  std::string language() const
  {
    return language_;
  }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<person>(*this);
    ar & language_;
  }

  std::string language_;
};

void save()
{
  boost::archive::text_oarchive oa(ss);
  oa.register_type<developer>();
  person *p = new developer(31, "C++");
  oa << p;
```

```
  delete p;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  ia.register_type<developer>();
  person *p;
  ia >> p;
  std::cout << p->age() << std::endl;
  delete p;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

上面的应用程序没有使用 `BOOST_CLASS_EXPORT` 宏，而是调用了 `register_type ()` 模板函数。 需要注册的类型作为模板参数传入。

请注意 `register_type ()` 必须在 `save ()` 和 `load ()` 都要调用。

`register_type ()` 的优点是只有需要序列化的类才注册。 比如在开发一个库时，你不知道开发人员将来要序列化哪些类。 当然 `BOOST_CLASS_EXPORT` 宏用起来简单，可它却可能注册那些不需要序列化的类型。

## 11.5. 优化用封装函数

在理解了如何序列化对象之后，本节介绍用来优化序列化过程的封装函数。 通过这个函数，对象被打上标记允许 Boost.Serialization 使用一些优化技术。

下面例子使用不带封装函数的 Boost.Serialization 。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/array.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

void save()
{
  boost::archive::text_oarchive oa(ss);
  boost::array<int, 3> a = { 0, 1, 2 };
  oa << a;
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  boost::array<int, 3> a;
  ia >> a;
  std::cout << a[0] << ", " << a[1] << ", " << a[2] << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

上面的应用程序创建一个文本流 `22 serialization::archive 5 0 0 3 0 1 2` 并将其写到标准输出流中。 使用封装函数 `boost::serialization::make_array()` ，输出可以缩短到 `22 serialization::archive 5 0 1 2` 。

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/array.hpp>
#include <boost/array.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

void save()
{
  boost::archive::text_oarchive oa(ss);
  boost::array<int, 3> a = { 0, 1, 2 };
  oa << boost::serialization::make_array(a.data(), a.size());
}

void load()
{
  boost::archive::text_iarchive ia(ss);
  boost::array<int, 3> a;
  ia >> boost::serialization::make_array(a.data(), a.size());
  std::cout << a[0] << ", " << a[1] << ", " << a[2] << std::endl;
}

int main()
{
  save();
  load();
}
```

- 下载源代码

`boost::serialization::make_array()` 函数需要地址和数组的长度。 由于长度是硬编码的，所以它不需要作为 `boost::array` 类型的一部分序列化。任何时候，如果 `boost::array` 或 `std::vector` 包含一个可以直接序列化的数组，都可以使用这个函数。 其他一般需要序列化的属性不能被序列化。

另一个 Boost.Serialization 提供的封装函数是 `boost::serialization::make_binary_object()` 。 与 `boost::serialization::make_array()` 类似，它也需要地址和长度。 `boost::serialization::make_binary_object()` 函数只是为了用来序列化没有底层结构的二进制数据，而 `boost::serialization::make_array()` 是用来序列化数组的。

## 11.6. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 开发一个应用程序，能够将任意个数有名称，部门和雇员唯一标识号构成的记录， 序列化到文件并从中恢复。 记录应该在恢复后在屏幕上显示。 用样本记录测试应用程序。

2. 扩展上面的应用程序，为每个雇员存储生日。 应用程序应该还可以恢复 在上面的练习创建的的旧版本的记录。

# 第 12 章 词法分析器

## 目录







## 12.1. 概述

词法分析器用于读取各种格式的数据，这些数据可以具有灵活但可能非常复杂的结构。 关于"格式"的一个最好的例子就是 C++ 代码。 编译器的词法分析器必须理解 C++ 的各种可能的语言结构组合，以将它们翻译为某种二进制形式。

开发词法分析器的主要问题是所分析的数据的组成结构具有大量的规则。 例如，C++ 支持很多的语言结构，开发一个相应的词法分析器可能需要无数个 `if` 表达式来识别任意所能想象到的 C++ 代码是否有效。

本章所介绍的 Boost.Spirit 库将词法分析器的开发放到了桌面上来。 无需将明确的规则转换为代码并使用无数的 `if` 表达式来验证代码，Boost.Spirit 可以使用一种被称为扩展BNF范式的东西来表示规则。 通过使用这些规则，Boost.Spirit 就可以对一个 C++ 源代码进行分析。

Boost.Spirit 的基本思想类似于正则表达式。 即不用 `if` 表达式来查找指定模式的文本，而是将模式以正则表达式的方式指定出来。 然后就可以使用象 Boost.Regex 这样的库来执行相应的查找国，因此开发者无需关心其中的细节。

本章将示范如何用 Boost.Spirit 来读入正则表达式不再适用的复杂格式。 由于 Boost.Spirit 是一个功能非常全的库，引入了多个不同的概念，所以在本章我们将开发一个 JSON 格式 的简单的词法分析器。 JSON 是被如 Ajax 一类的应用程序用于在程序之间交换数据的格式，类似于 XML，可以在不同平台上运行。

虽然 Boost.Spirit 简化了词法分析器的开发，但是还没有人能够成功地基于这个库写出一个 C++ 词法分析器。 这类词法分析器的开发仍然是 Boost.Spirit 的一个长期目标，不过，由于 C++ 语言的复杂性，目前还未能实现。 Boost.Spirit 目前还不能很好地适用于这种复杂性或二进制格式。

## 12.2. 扩展BNF范式

Backus-Naur 范式，简称 BNF，是一种精确描述规则的语言，被应用于多种技术规范。例如，众多互联网协议的许多技术规范，称为 RFC，除了文字说明以外，都包含了以 BNF 编写的规则。

Boost.Spirit 支持扩展BNF范式(EBNF)，可以用比 BNF 更简短的方式来指定规则。EBNF 的主要优点就是简短，从而写法更简单。

请注意，EBNF 有几种不同的变体，它们的语法可能有些差异。本章以及 Boost.Spirit 所使用的 EBNF 语法类似于正则表达式。

要使用 Boost.Spirit，你必须懂得 EBNF。多数情况下，开发者已经知道 EBNF，因此才会选择 Boost.Spirit 来重用以前用 EBNF 表示的规则。以下是对 EBNF 的一个简短介绍；如果需要对本章当中以及 Boost.Spirit 所使用的语法有一个快速的参考，请查看 W3C XML 规范，其中包含了一个 短摘要。

| `digit` | = | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |

严格地讲，EBNF 以生成规则来表示规则。可以将任意数量的生成规则组合起来，描述一个特定的格式。以上格式只包含一个生成规则。它定义了一个 `digit` 是由0至9之间的任一数字组成。

象 `digit` 这样的定义被称为非终结符号。以上定义中的数字 0 到 9 则被称为终结符号。这些符号不具有任意特定意义，而且很容易识别出来，因为它们是用双引号引起来的。

所有数字值是用竖直符相连的，其意义与 C++ 中的 `||` 操作符一样：多选一。

一句话，这个生成规则指明了0至9之间的任一数字都是一个 `digit` 。

| `integer` | = | ("+" | "-")? `digit` + |

这个新的非终结符 `integer` 包含至少一个 `digit` ，而且可选地以一个加号或减号开头。

`integer` 的定义用到了多个新的操作符。圆括号用于创建一个子表达式，就象它在数学中的作用。其它操作符可应用于这些子表达式。问号表示这个子表达式只能出现一次或不出现。

`digit` 之后的加号表示相应的表达式必须出现至少一次。

这个新的生成规则定义了一个任意的正或负的整数。一个 `digit` 正好是一个数字，而一个 `integer` 则可以由多个数字组成，且可以被标记为无符号的或有符号的。因此 5 即是一个 `digit` 也是一个 `integer` ，而 +5 则只是一个 `integer` 。同样地，169 或 -8 也只是 `integer` 。

通过定义和组合各种非终结符，可以创建越来越复杂的生成规则。

| `real` | = | `integer` "." `digit` * |

`integer` 的定义表示的是整数，而 `real` 的定义则表示了浮点数。这个规则基于前面已定义的非终结符 `integer` 和 `digit` ，以一个句点号分隔。 `digit` 之后的星号表示点号之后的数字是可选的：可以有任意多个数字或没有数字。

浮点数如 1.2, -16.99 甚至 3. 都符合 `real` 的定义。 但是，当前的定义不允许浮点数不带前导的零，如 .9。

正如本章开始的时候所提到的，接下来我们要用 Boost.Spirit 开发一个 JSON 格式的词法分析器。 为此，需要用 EBNF 来给出 JSON 格式的规则。

| `object` | = | "{" `member` ("," `member` ) *"}"* | | *`member`* | = | *`string`* *":"* *`value`* | | *`string`* | = | *'"'* *`character`* '"' | | `value` | = | `string` | `number` | `object` | `array` | "true" | "false" | "null" | | `number` | = | `integer` | `real` | | `array` | = | "[" `value` ("," `value` )* "]" | | `character` | = | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |

JSON 格式基于一些包含了键值和值的成对的对象，它们被花括号括起来。 其中键值是普通的字符串，而值可以是字符串、数字值、数组、其它对象或是字面值 `true`，`false` 或 `null`。 字符串是由双引号引起来的连续字符。 数字值可以是整数或浮点数。 数组包含以逗号分隔的值，并以方括号括起来。

请注意，以上定义并不完整。 一方面， `character` 的定义缺少了大写字母以及其它字符；另一方面，JSON 还特别支持 Unicode 或控制字符。 这些现在都可以先忽略掉，因为 Boost.Spirit 定义了常用的非终结符号，如字母数字字符，以减少你打字的次数。 另外，稍后在代码中，字符串被定义为除双引号以外的任意字符的连续串。 由于双引号用于结束一个字符串，所以其它所有字符都在字符串中使用。 上述 EBNF 并不如此表示，因为 EBNF 要求定义一个包含除单个字符外的所有字符的非终结符号，应该定义一个例外来排除。

以下是使用了上述定义的 JSON 格式的一个例子。

```
{
  "Boris Schäling" :
  {
    "Male": true,
    "Programming Languages": [ "C++", "Java", "C#" ],
    "Age": 31
  }
}
```

整个对象由最外层的花括号给出，它包含了一个键-值对。 键值是 "Boris Schäling"，值则是一个新的对象，包含多个键-值对。 其中所有键值均为字符串，而值则分别为字面值 `true`，一个包含几个字符串的数组，以及一个数字值。

以上所定义的 EBNF 规则现在就可用于通过 Boost.Spirit 开发一个可以读取以上 JSON 格式的词法分析器。

## 12.3. 语法

继上一节中以 EBNF 为 JSON 格式定义了相关规则后，现在要将这些规则与 Boost.Spirit 一起使用。 Boost.Spirit 实际上允许以 C++ 代码来定义 EBNF 规则，方法是重载各个由 EBNF 使用的不同操作符。

请注意，EBNF 规则需要稍作修改，才能创建出合法的 C++ 代码。 在 EBNF 中各个符号是以空格相连的，在 C++ 中需要用某个操作符来连接。 此外，象星号、问号和加号这些操作符，在 EBNF 中是置于对应的符号后面的，在 C++ 中必须置于符号的前面，才能作为单参操作符来使用。

以下是在 Boost.Spirit 中为表示 JSON 格式，用 C++ 代码写的 EBNF 规则。

```
#include <boost/spirit.hpp>

struct json_grammar
  : public boost::spirit::grammar<json_grammar>
{
  template <typename Scanner>
  struct definition
  {
    boost::spirit::rule<Scanner> object, member, string, value, nur

    definition(const json_grammar &self)
    {
      using namespace boost::spirit;
      object = "{" >> member >> *("," >> member) >> "}";
      member = string >> ":" >> value;
      string = "\"" >> *~ch_p("\"") >> "\"";
      value = string | number | object | array | "true" | "false"
      number = real_p;
      array = "[" >> value >> *("," >> value) >> "]";
    }

    const boost::spirit::rule<Scanner> &start()
    {
      return object;
    }
  };
};

int main()
{
}
```

- 下载源代码

为了使用 Boost.Spirit 中的各个类，需要包含头文件 `boost/spirit.hpp` 。 所有类均位于名字空间 `boost::spirit` 内。

为了用 Boost.Spirit 创建一个词法分析器，除了那些定义了数据是如何构成的规则以外，还必须创建一个所谓的语法。 在上例中，就创建一个 `json_grammar` 类，它派生自模板类 `boost::spirit::grammar`，并以该类的名字来实例化。 `json_grammar` 定义了理解 JSON 格式所需的完整语法。

语法的一个最重要的组成部分就是正确读入结构化数据的规则。 这些规则在一个名为 `definition` 的内层类中定义 - 这个名字是强制性的。 这个类是带有一个模板参数的模板类，由 Boost.Spirit 以一个所谓的扫描器来进行实例化。 扫描器是 Boost.Spirit 内部使用的一个概念。 虽然强制要求 `definition` 必须是以一个扫描器类型作为其模板参数的模板类，但是对于 Boost.Spirit 的日常使用来说，这些扫描器是什么以及为什么要定义它们，并不重要。

`definition` 必须定义一个名为 `start()` 的方法，它会被 Boost.Spirit 调用，以获得该语法的完整规则和标准。 这个方法的返回值是 `boost::spirit::rule` 的一个常量引用，它也是一个以扫描器类型实例化的模板类。

`boost::spirit::rule` 类用于定义规则。 非终结符号就以这个类来定义。 前面所定义的非终结符号 `object`，`member`，`string`，`value`，`number` 和 `array` 的类型均为 `boost::spirit::rule`。

所有这些对象都被定义为 `definition` 类的属性，这并非强制性的，但简化了定义，尤其是当各个规则之间有递归引用时。 正如在上一节中所看到的 EBNF 例子那样，递归引用并不是一个问题。

乍一看，在 `definition` 的构造函数内的规则定义非常类似于在上一节中看到的 EBNF 生成规则。 这并不奇怪，因为这正是 Boost.Spirit 的目标：重用在 EBNF 中定义的生成规则。

由于是用 C++ 代码来组成 EBNF 中建立的规则，为了写出合法的 C++，其实是有一点点差异的。 例如，所有符号间的连接是通过 `>>` 操作符完成的。 EBNF 中的一些操作符，如星号，被置于相应符号的前面而非后面。 尽管有这样一些语法上的修改，Boost.Spirit 还是尽量在将 EBNF 规则转换至 C++ 代码时不进行太多的修改。

`definition` 的构造函数使用了由 Boost.Spirit 提供的两个类：`boost::spirit::ch_p` 和 `boost::spirit::real_p`。 这些以分析器形式提供的常用规则可以很方便地重用。 一个例子就是 `boost::spirit::real_p`，它可以用于保存正或负的整数或浮点数，无需定义象 `digit` 或 `real` 这样的非终结符号。

`boost::spirit::ch_p` 可用于创建一个针对单个字符的分析器，相当于将字符置于双引号中。 在上例中， `boost::spirit::ch_p` 的使用是强制性的，因为波浪号和星号是要应用于双引号之上的。 没有这个类，代码将变为 `*~"\""`，这会被编译器拒绝为非法代码。

波浪号实际上是实现了前一节中提到的一个技巧：在双引号之前加上波浪号，可以接受除双引号以外的所有其它字符。

定义完了识别 JSON 格式的规则后，以下例子示范了如何使用这些规则。

```
#include <boost/spirit.hpp>
#include <fstream>
#include <sstream>
#include <iostream>

struct json_grammar
  : public boost::spirit::grammar<json_grammar>
{
  template <typename Scanner>
  struct definition
  {
    boost::spirit::rule<Scanner> object, member, string, value, nur

    definition(const json_grammar &self)
    {
      using namespace boost::spirit;
      object = "{" >> member >> *("," >> member) >> "}";
      member = string >> ":" >> value;
      string = "\"" >> *~ch_p("\"") >> "\"";
      value = string | number | object | array | "true" | "false"
      number = real_p;
      array = "[" >> value >> *("," >> value) >> "]";
    }

    const boost::spirit::rule<Scanner> &start()
    {
      return object;
    }
  };
};

int main(int argc, char *argv[])
{
  std::ifstream fs(argv[1]);
  std::ostringstream ss;
  ss << fs.rdbuf();
  std::string data = ss.str();

  json_grammar g;
  boost::spirit::parse_info<> pi = boost::spirit::parse(data.c_str(
  if (pi.hit)
  {
    if (pi.full)
      std::cout << "parsing all data successfully" << std::endl;
    else
      std::cout << "parsing data partially" << std::endl;
    std::cout << pi.length << " characters parsed" << std::endl;
  }
  else
    std::cout << "parsing failed; stopped at '" << pi.stop << "'" <
}
```

- 下载源代码

Boost.Spirit 提供了一个名为 `boost::spirit::parse()` 的自由函数。 通过创建一个语法的实例，就会相应地创建一个词法分析器，该分析器被作为第二个参数传递给 `boost::spirit::parse()` 。 第一个参数表示要进行分析的文本，而第三个参数则是一个表明在给定文本中哪些字符将被跳过的词法分析器。 为了跳过空格，要将一个类型为 `boost::spirit::space_p` 的对象作为第三个参数传入。 这只是表示在被捕获的数据之间 - 换句话说，就是规则中使用了 `>>` 操作符的地方 - 可以有任意数量的空格。 这其中包含了制表符和换行符，令数据的格式可以更为灵活。

`boost::spirit::parse()` 返回一个类型为 `boost::spirit::parse_info` 的对象，该对象提供了四个属性来表示文本是否被成功分析。 如果文本被成功分析，则属性 `hit` 被设置为 `true` 。 如果文本中的所有字符都被分析完了，最后没有剩余空格，则 `full` 也被设置为 `true` 。 仅当 `hit` 为 `true` 时， `length` 是有效的，其中保存了被成功分析的字符数量。

如果文本未能分析成功，则属性 `length` 不能被访问。 此时，可以访问属性 `stop` 来获得停止分析的文本位置。 如果文本被成功分析， `stop` 也是可访问的，只不过没什么意义，因为此时它肯定是指向被分析文本之后。

## 12.4. 动作

到目前为止，你已经知道了如何定义一个语法，以得到一个新的词法分析器，用于识别一个给定的文本是否具有该语法的规则所规定的结构。 但是此刻，数据的格式仍未被解释，因为从结构化格式如 JSON 中所读取的数据并没有被进一步处理。

要对由分析器识别出来的符合某个特定规则的数据进行处理，可以使用动作 (action)。 动作是一些与规则相关联的函数。 如果词法分析器识别出某些数据符合某个特定的规则，则相关联的动作会被执行，并把识别得到的数据传入进行处理，如下例所示。

```
#include <boost/spirit.hpp>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

struct json_grammar
  : public boost::spirit::grammar<json_grammar>
{
  struct print
  {
    void operator()(const char *begin, const char *end) const
    {
      std::cout << std::string(begin, end) << std::endl;
    }
  };
```

```
  template <typename Scanner>
  struct definition
  {
    boost::spirit::rule<Scanner> object, member, string, value, nur

    definition(const json_grammar &self)
    {
      using namespace boost::spirit;
      object = "{" >> member >> *("," >> member) >> "}";
      member = string[print()] >> ":" >> value;
      string = "\"" >> *~ch_p("\"") >> "\"";
      value = string | number | object | array | "true" | "false"
      number = real_p;
      array = "[" >> value >> *("," >> value) >> "]";
    }

    const boost::spirit::rule<Scanner> &start()
    {
      return object;
    }
  };
};

int main(int argc, char *argv[])
{
  std::ifstream fs(argv[1]);
  std::ostringstream ss;
  ss << fs.rdbuf();
  std::string data = ss.str();

  json_grammar g;
  boost::spirit::parse_info<> pi = boost::spirit::parse(data.c_str(
  if (pi.hit)
  {
    if (pi.full)
      std::cout << "parsing all data successfully" << std::endl;
    else
      std::cout << "parsing data partially" << std::endl;
    std::cout << pi.length << " characters parsed" << std::endl;
  }
  else
    std::cout << "parsing failed; stopped at '" << pi.stop << "'" <
}
```

- 下载源代码

动作被实现为函数或函数对象。 如果动作需要被初始化或是要在多次执行之间维护某些状态信息，则后者更好一些。 以上例子中将动作实现为函数对象。

类 `print` 是一个函数对象，它将数据写出至标准输出流。 当其被调用时，重载的 `operator()()` 操作符将接受一对指向数据起始点和结束点的指针，所指范围即为被执行该动作的规则所识别出来的数据。

这个例子将这个动作关联至在 `member` 之后作为第一个符号出现的非终结符号 `string` 。 一个类型为 `print` 的实例被放在方括号内传递给非终结符号 `string` 。 由于 `string` 表示的是 JSON 对象的键-值对中的键，所以每次找到一个键时，类 `print` 中的重载 `operator()()` 操作符将被调用，将该键写出到标准输出流。

我们可以定义任意数量的动作，或将它们关联至任意数量的符号。 要把一个动作关联至一个字面值，必须明确给出一个词法分析器。 这与在非终结符号 `string` 的定义中指定 `boost::spirit::ch_p` 类没什么不同。 以下例子使用了 `boost::spirit::str_p` 类来将一个 `print` 类型的对象关联至字面值 `true` 。

```
#include <boost/spirit.hpp>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

struct json_grammar
  : public boost::spirit::grammar<json_grammar>
{
  struct print
  {
    void operator()(const char *begin, const char *end) const
    {
      std::cout << std::string(begin, end) << std::endl;
    }

    void operator()(const double d) const
    {
      std::cout << d << std::endl;
    }
  };

  template <typename Scanner>
  struct definition
  {
    boost::spirit::rule<Scanner> object, member, string, value, nur

    definition(const json_grammar &self)
    {
      using namespace boost::spirit;
      object = "{" >> member >> *("," >> member) >> "}";
      member = string[print()] >> ":" >> value;
      string = "\"" >> *~ch_p("\"") >> "\"";
      value = string | number | object | array | str_p("true")[prir
      number = real_p[print()];
```

```
      array = "[" >> value >> *("," >> value) >> "]";
    }

    const boost::spirit::rule<Scanner> &start()
    {
      return object;
    }
  };
};

int main(int argc, char *argv[])
{
  std::ifstream fs(argv[1]);
  std::ostringstream ss;
  ss << fs.rdbuf();
  std::string data = ss.str();

  json_grammar g;
  boost::spirit::parse_info<> pi = boost::spirit::parse(data.c_str(
  if (pi.hit)
  {
    if (pi.full)
      std::cout << "parsing all data successfully" << std::endl;
    else
      std::cout << "parsing data partially" << std::endl;
    std::cout << pi.length << " characters parsed" << std::endl;
  }
  else
    std::cout << "parsing failed; stopped at '" << pi.stop << "'" <
}
```

- 下载源代码

另外，这个例子还将一个动作关联至 `boost::spirit::real_p` 。 大多数分析器会传递一对指向被识别数据起始点和结束点的指针，而 `boost::spirit::real_p` 则将所找到的数字作为 `double` 来传递。 这样可以使对数字的处理更为方便，因为这些数字不再需要被显式转换。 为了传递一个 `double` 类型的值给这个动作，我们相应地增加了一个重载的 `operator()()` 操作符给 `print` 。

除了在本章中介绍过的分析器，如 `boost::spirit::str_p` 或 `boost::spirit::real_p` 以外，Boost.Spirit 还提供了很多其它的分析器。 例如，如果要使用正则表达式，我们有 `boost::spirit::regex_p` 可用。 此外，还有用于验证条件或执行循环的分析器。 它们有助于创建动态的词法分析器，根据条件来对数据进行不同的处理。 要对 Boost.Spirit 提供的这些工具有一个大概的了解，你应该看一下这个库的文档。

## 12.5. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 开发一个可以对任意整数和浮点数进行加减的计算器。 这个计算器接受形如 **`=-4+8 + 1.5`** 的输入，并显示结果 `5.5` 。

# 第 13 章 容器

目录







## 13.1. 概述

这一章将会介绍许多我们在 C++ 标准中已经很熟悉的容器的 Boost 版本。 在这一章里， 我们会对 Boost.Unordered 的用法有一定的了解 （这个容器已经在 TR1 里被加入到了 C++ 标准）； 我们将会学习如何定义一个 Boost.MultiIndex； 我们还会了解何时应该使用 MuitiIndex 的一个特殊的扩展 —— Boost.Bimap。 接下来，我们会向你介绍第一个容器 —— Boost.Array， 通过使用它， 你可以把 C++ 标准里普通的数组以容器的形式实现。

## 13.2. Boost.Array

库 Boost.Array 在 `boost/array.hpp` 中定义了一个模板类 `boost::array` 。通过使用这个类， 你可以创建一个跟 C++ 里传统的数组有着相同属性的容器。 而且， `boost::array` 还满足了 C++ 中容器的一切需求， 因此， 你可以像操作容器一样方便的操作这个 array。 基本上， 你可以把 `boost::array` 当成 `std::vector` 来使用， 只不过 `boost::array` 是定长的。

```
#include <boost/array.hpp>
#include <iostream>
#include <string>
#include <algorithm>

int main()
{
  typedef boost::array<std::string, 3> array;
  array a;

  a[0] = "Boris";
  a.at(1) = "Anton";
  *a.rbegin() = "Caesar";

  std::sort(a.begin(), a.end());

  for (array::const_iterator it = a.begin(); it != a.end(); ++it)
    std::cout << *it << std::endl;

  std::cout << a.size() << std::endl;
  std::cout << a.max_size() << std::endl;
}
```

- 下载源代码

就像我们在上面的例子看到的那样， `boost::array` 简直就是简单的不需要任何多余的解释， 因为所有操作都跟 `std::vector` 是一样的。

在下面的例子里， 我们会见识到 Boost.Array 的一个特性。

```
#include <boost/array.hpp>
#include <string>

int main()
{
  typedef boost::array<std::string, 3> array;
  array a = { "Boris", "Anton", "Caesar" };
}
```

- 下载源代码

一个 `boost::array` 类型的数组可以使用传统 C++ 数组的初始化方式来初始化。

既然这个容器已经在 TR1 中加入到了 C++ 标准， 它同样可以通过 `std::array` 来访问到。 他被定义在头文件 `array` 中， 使用它的前提是你正在使用一个支持 TR1 的 C++ 标准的库。

# 13.3. Boost.Unordered

Boost.Unordered 在 C++ 标准容器 `std::set` ， `std::multiset` ， `std::map` 和 `std::multimap` 的基础上多实现了四个容器： `boost::unordered_set` ， `boost::unordered_multiset` ， `boost::unordered_map` 和 `boost::unordered_multimap` 。 那些名字很相似的容器之间并没有什么不同， 甚至还提供了相同的接口。 在很多情况下， 替换这两种容器 (std 和 boost) 对你的应用不会造成任何影响。

Boost.Unordered 和 C++ 标准里的容器的不同之处在于—— Boost.Unordered 不要求其中的元素是可排序的， 因为它不会做出排序操作。 在排序操作无足轻重时（或是根本不需要）， Boost.Unordered 就很合适了。

为了能够快速的查找元素， 我们需要使用 Hash 值。 Hash 值是一些可以唯一标识容器中元素的数字， 它在比较时比起类似 String 的数据类型会更加有效率。 为了计算 Hash 值， 容器中的所有元素都必须支持对他们自己唯一 ID 的计算。 比如 `std::set` 要求其中的元素都要是可比较的， 而 `boost::unordered_set` 要求其中的元素都要可计算 Hash 值。 尽管如此， 在对排序没有需求时， 你还是应该倾向使用 Boost.Unordered 。

下面的例子展示了 `boost::unordered_set` 的用法。

```
#include <boost/unordered_set.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::unordered_set<std::string> unordered_set;
  unordered_set set;

  set.insert("Boris");
  set.insert("Anton");
  set.insert("Caesar");

  for (unordered_set::iterator it = set.begin(); it != set.end(); -
    std::cout << *it << std::endl;

  std::cout << set.size() << std::endl;
  std::cout << set.max_size() << std::endl;

  std::cout << (set.find("David") != set.end()) << std::endl;
  std::cout << set.count("Boris") << std::endl;
}
```

- 下载源代码

`boost::unordered_set` 提供了与 `std::set` 相似的函数。 当然， 这个例子不需要多大改进就可以用 `std::set` 来实现。

下面的例子展示了如何用 `boost::unordered_map` 来存储每一个的 person 的 name 和 age。

```
#include <boost/unordered_map.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::unordered_map<std::string, int> unordered_map;
  unordered_map map;

  map.insert(unordered_map::value_type("Boris", 31));
  map.insert(unordered_map::value_type("Anton", 35));
  map.insert(unordered_map::value_type("Caesar", 25));

  for (unordered_map::iterator it = map.begin(); it != map.end(); -
    std::cout << it->first << ", " << it->second << std::endl;

  std::cout << map.size() << std::endl;
  std::cout << map.max_size() << std::endl;

  std::cout << (map.find("David") != map.end()) << std::endl;
  std::cout << map.count("Boris") << std::endl;
}
```

- 下载源代码

就像我们看到的， `boost::unordered_map` 和 `std::map` 之间并没多大区别。 同样地， 你可以很方便的用 `std::map` 来重新实现这个例子。

就像上面提到过的， Boost.Unordered 需要其中的元素可计算 Hash 值。 一些类似于 `std::string` 的数据类型“天生”就支持 Hash 值的计算。 对于那些自定义的类型， 你需要手动的定义 Hash 函数。

```
#include <boost/unordered_set.hpp>
#include <string>

struct person
{
  std::string name;
  int age;

  person(const std::string &n, int a)
    : name(n), age(a)
  {
  }

  bool operator==(const person &p) const
  {
    return name == p.name && age == p.age;
  }
};

std::size_t hash_value(person const &p)
{
  std::size_t seed = 0;
  boost::hash_combine(seed, p.name);
  boost::hash_combine(seed, p.age);
  return seed;
}

int main()
{
  typedef boost::unordered_set<person> unordered_set;
  unordered_set set;

  set.insert(person("Boris", 31));
  set.insert(person("Anton", 35));
  set.insert(person("Caesar", 25));
}
```

- 下载源代码

在代码中， `person` 类型的元素被存到了 `boost::unordered_set` 中。 因为 `boost::unordered_set` 中的 Hash 函数不能识别 `person` 类型， Hash 值就变得无法计算了。 若果没有定义另一个 Hash 函数， 你的代码将不会通过编译。

Hash 函数的签名必须是： `hash_value()` 。 它接受唯一的一个参数来指明需要计算 Hash 值的对象的类型。 因为 Hash 值是单纯的数字， 所以函数的返回值为： `std::size_t` 。

每当一个对象需要计算它的 Hash 值时， `hash_value()` 都会自动被调用。 Boost C++ 库已经为一些数据类型定义好了 Hash 函数， 比如： `std::string` 。 但对于像 `person` 这样的自定义类型， 你就需要自己手工定义了。

`hash_value()` 的实现往往都很简单：你只需要按顺序对其中的每个属性都调用 Boost 在 `boost/functional/hash.hpp` 中提供的 `boost::hash_combine()` 函数就行了。当你使用 Boost.Unordered 时，这个头文件已经自动被包含了。

除了自定义 `hash_value()` 函数，自定义的类型还需要支持通过 `==` 运算符的比较操作。因此，`person` 就重载了相应的 `operator==()` 操作符。

## 13.4. Boost.MultiIndex

Boost.MultiIndex 比我们之前介绍的任何库都要复杂。不像 Boost.Array 和 Boost.Unordered 为我们提供了可以直接使用的容器，Boost.MultiIndex 让我们可以自定义新的容器。跟 C++ 标准中的容器不同的是：一个用户自定义的容器可以对其中的数据提供多组访问接口。举例来说，你可以定义一个类似于 `std::map` 的容器，但它可以通过 value 值来查询。如果不用 Boost.MultiIndex，你就需要自己整合两个 `std::map` 类型的容器，还要自己处理一些同步操作来确保数据的完整性。

下面这个例子就用 Boost.MultiIndex 定义了一个新容器来存储每个人的 name 和 age，不像 `std::map`，这个容器可以分别通过 name 和 age 来查询（std::map 只能用一个值）。

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <iostream>
#include <string>

struct person
{
  std::string name;
  int age;

  person(const std::string &n, int a)
    : name(n), age(a)
  {
  }
};

typedef boost::multi_index::multi_index_container<
  person,
  boost::multi_index::indexed_by<
    boost::multi_index::hashed_non_unique<
      boost::multi_index::member<
        person, std::string, &person::name
      >
    >,
    boost::multi_index::hashed_non_unique<
      boost::multi_index::member<
        person, int, &person::age
      >
    >
  >
> person_multi;

int main()
{
  person_multi persons;

  persons.insert(person("Boris", 31));
  persons.insert(person("Anton", 35));
  persons.insert(person("Caesar", 25));

  std::cout << persons.count("Boris") << std::endl;

  const person_multi::nth_index<1>::type &age_index = persons.get<
  std::cout << age_index.count(25) << std::endl;
}
```

- 下载源代码

就像上面提到的， Boost.MultiIndex 并没有提供任何特定的容器而是一些类来方便我们定义新的容器。 典型的做法是： 你需要用到 `typedef` 来为你的新容器提供对 Boost.MultiIndex 中类的方便的访问。

每个容器定义都需要的类 `boost::multi_index::multi_index_container` 被定义在了 `boost/multi_index_container.hpp` 里。 因为他是一个模板类， 你需要为它传递两个模板参数。 第一个参数是容器中储存的元素类型， 在例子中是 `person` ； 而第二个参数指明了容器所提供的所有索引类型。

基于 Boost.MultiIndex 的容器最大的优势在于： 他对一组同样的数据提供了多组访问接口。 访问接口的具体细节都可以在定义容器时被指定。 因为例子中的 person 为 age 和 name 都提供了查询功能， 我们必须要定义两组接口。

接口的定义必须借由模板类 `boost::multi_index::indexed_by` 来实现。 每一个接口都作为参数传递给它。 例子中定义了两个 `boost::multi_index::hashed_non_unique` 类型的接口， （定义在头文件 `boost/multi_index/hashed_index.hpp` 中） 如果你希望容器像 Boost.Unordered 一样存储一些可以计算 Hash 值的元素， 你就可以使用这个接口。

`boost::multi_index::hashed_non_unique` 是一个模板类， 他需要一个可计算 Hash 值的类型作为它的参数。 因为接口需要访问 person 中的 name 和 age， 所以 name 和 age 都要是可计算 Hash 值的。

Boost.MultiIndex 提供了一个辅助模板类： `boost::multi_index::member` （定义在 `boost/multi_index/member.hpp` 中） 来访问类中的属性。 就像我们在例子中所看到的， 我们指定了好几个参数来让 `boost::multi_index::member` 明白可以访问 `person` 中的哪些属性以及这些属性的类型。

不得不说 `person_multi` 的定义第一眼看起来相当复杂， 但这个类本身跟 Boost.Unordered 中的 `boost::unordered_map` 并没有什么不同， 他也可以分别通过其中的两个属性 name 和 age 来查询容器。

为了访问 MultiIndex 容器， 你必须要定义至少一个接口。 如果用 `insert()` 或者 `count()` 来直接访问 `persons` 对象， 第一个接口会被隐式的调用 —— 在例子中是 `name` 属性的 Hash 容器。 如果你想用其他接口， 你必须要显示的指定它。

接口都是用从0开始的索引值来编号的。 想要访问第二个接口， 你需要调用 `get()` 函数并且传入想要访问的接口的索引值。

函数 `get()` 的返回值看起来也有点复杂： 他是一个用来访问 MultiIndex 容器的类 `nth_index` ， 同样的， 你也需要指定需要访问的接口的索引值。 当然， 这个值肯定跟 `get()` 函数指定的模板参数是一样的。 最后一步： 用 `::` 来得到 `nth_index` 的 `type` ， 也就是接口的真正的 `type` 。

虽然我们并不知道细节就用 `nth_index` 和 `type` 得到了接口， 我们还是需要明白这到底是什么接口。 通过传给 `get()` 和 `nth_index` 的索引值， 我们就可以很容易得知所使用的哪一个接口了。 例子中的 `age_index` 就是一个通过 age 来访问的 Hash 容器。

既然 MultiIndex 容器中的 name 和 key 作为了接口访问的键值， 他们都不能再被更改了。 比如一个 person 的 age 在通过 name 搜索以后被改变了， 使用 age 作为键值的接口却意识不到这种更改， 因此， 你需要重新计算 Hash 值才行。

就像 `std::map` 一样， MultiIndex 容器中的值也不允许被修改。 严格的说， 所有存储在 MultiIndex 中的元素都该是常量。 为了避免删除或修改其中元素真正的值， Boost.MultiIndex 提供了一些常用函数来操作其中的元素。 使用这些函数来操作 MultiIndex 容器中的值并不会引起那些元素所指向的真正的对象改变， 所以更新动作是安全的。 而且所有接口都会被通知这种改变， 然后去重新计算新的 Hash 值等。

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <iostream>
#include <string>

struct person
{
  std::string name;
  int age;

  person(const std::string &n, int a)
    : name(n), age(a)
  {
  }
};

typedef boost::multi_index::multi_index_container<
  person,
  boost::multi_index::indexed_by<
    boost::multi_index::hashed_non_unique<
      boost::multi_index::member<
        person, std::string, &person::name
      >
    >,
    boost::multi_index::hashed_non_unique<
      boost::multi_index::member<
        person, int, &person::age
      >
    >
  >
> person_multi;

void set_age(person &p)
{
  p.age = 32;
}

int main()
{
  person_multi persons;
```

```
  persons.insert(person("Boris", 31));
  persons.insert(person("Anton", 35));
  persons.insert(person("Caesar", 25));

  person_multi::iterator it = persons.find("Boris");
  persons.modify(it, set_age);

  const person_multi::nth_index<1>::type &age_index = persons.get<1
  std::cout << age_index.count(32) << std::endl;
}
```

- 下载源代码

每个 Boost.MultiIndex 中的接口都支持 `modify()` 函数来提供直接对容器本身的操作。 它的第一个参数是一个需要更改对象的迭代器； 第二参数则是一个对该对象进行操作的函数。 在例子中， 对应的两个参数则是： `person` 和 `set_age()` 。

至此， 我们都还只介绍了一个接口： `boost::multi_index::hashed_non_unique` ， 他会计算其中元素的 Hash 值， 但并不要求是唯一的。 为了确保容器中存储的值是唯一的， 你可以使用 `boost::multi_index::hashed_unique` 接口。 请注意： 所有要被存入容器中的值都必须满足它的接口的限定。 只要一个接口限定了容器中的值必须是唯一的， 那其他接口都不会对该限定造成影响。

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <iostream>
#include <string>

struct person
{
  std::string name;
  int age;

  person(const std::string &n, int a)
    : name(n), age(a)
  {
  }
};

typedef boost::multi_index::multi_index_container<
  person,
  boost::multi_index::indexed_by<
    boost::multi_index::hashed_non_unique<
      boost::multi_index::member<
        person, std::string, &person::name
      >
    >,
    boost::multi_index::hashed_unique<
      boost::multi_index::member<
        person, int, &person::age
      >
    >
  >
> person_multi;

int main()
{
  person_multi persons;

  persons.insert(person("Boris", 31));
  persons.insert(person("Anton", 31));
  persons.insert(person("Caesar", 25));

  const person_multi::nth_index<1>::type &age_index = persons.get<1
  std::cout << age_index.count(31) << std::endl;
}
```

- 下载源代码

上例中的容器现在使用了 `boost::multi_index::hashed_unique` 来作为他的第二个接口， 因此他不允许其中有两个同 age 的 person 存在。

上面的代码尝试存储一个与 Boris 同 age 的 Anton， 因为这个动作违反了容器第二个接口的限定， 它（Anton）将不会被存入到容器中。 因此， 程序将会输出： `1` 而不是2。

接下来的例子向我们展示了 Boost.MultiIndex 中剩下的三个接口： `boost::multi_index::sequenced` ， `boost::multi_index::ordered_non_unique` 和 `boost::multi_index::random_access` 。

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/random_access_index.hpp>
#include <boost/multi_index/member.hpp>
#include <iostream>
#include <string>

struct person
{
  std::string name;
  int age;

  person(const std::string &n, int a)
    : name(n), age(a)
  {
  }
};

typedef boost::multi_index::multi_index_container<
  person,
  boost::multi_index::indexed_by<
    boost::multi_index::sequenced<>,
    boost::multi_index::ordered_non_unique<
      boost::multi_index::member<
        person, int, &person::age
      >
    >,
    boost::multi_index::random_access<>
  >
> person_multi;

int main()
{
  person_multi persons;

  persons.push_back(person("Boris", 31));
  persons.push_back(person("Anton", 31));
  persons.push_back(person("Caesar", 25));

  const person_multi::nth_index<1>::type &ordered_index = persons.g
  person_multi::nth_index<1>::type::iterator lower = ordered_index.
  person_multi::nth_index<1>::type::iterator upper = ordered_index.
  for (; lower != upper; ++lower)
    std::cout << lower->name << std::endl;

  const person_multi::nth_index<2>::type &random_access_index = per
  std::cout << random_access_index[2].name << std::endl;
}
```

- 下载源代码

`boost::multi_index::sequenced` 接口让我们可以像使用 `std::list` 一样的使用 MultiIndex。 这个接口定义起来十分容易： 你不用为它传递任何模板参数。 `person` 类型的对象在容器中就是像 list 一样按照加入的顺序来排列的。

而通过使用 `boost::multi_index::ordered_non_unique` 接口， 容器中的对象会自动的排序。 你在定义容器时就必须指定接口的排序规则。 示例中的对象 `person` 就是以 age 来排序的， 它借助了辅助类 `boost::multi_index::member` 来实现这一功能。

`boost::multi_index::ordered_non_unique` 为我们提供了一些特别的函数来查找特定范围的数据。 通过使用 `lower_bound()` 和 `upper_bound()` ， 示例实现了对所有 30 岁至 40 岁的 person 的查询。 要注意因为容器中的数据是有序的， 所以才提供了这些函数， 其他接口中并不提供这些函数。

最后一个接口是： `boost::multi_index::random_access` ， 他让我们可以像使用 `std::vector` 一样使用 MultiIndex 容器。 你又可以使用你熟悉的 `operator[]()` 和 `at()` 操作了。

请注意 `boost::multi_index::random_access` 已经被完整的包含在了 `boost::multi_index::sequenced` 接口中。 所以当你使用 `boost::multi_index::random_access` 的时候， 你也可以使用 `boost::multi_index::sequenced` 接口中的所有函数。

在介绍完 Boost.MultiIndex 剩下的4个接口后， 本章剩下的部分将向你介绍所谓的“键值提取器”（key extractors）。 目前为止， 我们已经见过一个在 `boost/multi_index/member.hpp` 定义的键值提取器了—— `boost::multi_index::member` 。 这个辅助函数的得名源自它可以显示的声明类中的哪些属性会作为接口中的键值使用。

接下来的例子介绍了另外两个键值提取器。

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/mem_fun.hpp>
#include <iostream>
#include <string>

class person
{
public:
  person(const std::string &n, int a)
    : name(n), age(a)
  {
  }

  bool operator<(const person &p) const
  {
```

```
    return age < p.age;
  }

  std::string get_name() const
  {
    return name;
  }

private:
  std::string name;
  int age;
};

typedef boost::multi_index::multi_index_container<
  person,
  boost::multi_index::indexed_by<
    boost::multi_index::ordered_unique<
      boost::multi_index::identity<person>
    >,
    boost::multi_index::hashed_unique<
      boost::multi_index::const_mem_fun<
        person, std::string, &person::get_name
      >
    >
  >
> person_multi;

int main()
{
  person_multi persons;

  persons.insert(person("Boris", 31));
  persons.insert(person("Anton", 31));
  persons.insert(person("Caesar", 25));

  std::cout << persons.begin()->get_name() << std::endl;

  const person_multi::nth_index<1>::type &hashed_index = persons.ge
  std::cout << hashed_index.count("Boris") << std::endl;
}
```

- 下载源代码

键值提取器 `boost::multi_index::identity` （定义在 `boost/multi_index/identity.hpp` 中） 可以使用容器中的数据类型作为键值。 示例中， 就需要 `person` 类是可排序的， 因为它已经作为了接口 `boost::multi_index::ordered_unique` 的键值。 在示例里， 它是通过重载 `operator<()` 操作符来实现的。

头文件 `boost/multi_index/mem_fun.hpp` 定义了两个可以把函数返回值作为键值的键值提取器： `boost::multi_index::const_mem_fun` 和 `boost::multi_index::mem_fun` 。在示例程序中，就是用到了 `get_name()` 的返回值作为键值。显而易见的， `boost::multi_index::const_mem_fun` 适用于返回常量的函数，而 `boost::multi_index::mem_fun` 适用于返回非常量的函数。

Boost.MultiIndex 还提供了两个键值提取器： `boost::multi_index::global_fun` 和 `boost::multi_index::composite_key` 。前一个适用于独立的函数或者静态函数，后一个允许你将几个键值提取器组合成一个新的的键值提取器。

# 13.5. Boost.Bimap

Boost.Bimap 库提供了一个建立在 Boost.MultiIndex 之上但不需要预先定义就可以使用的容器。这个容器十分类似于 `std::map` ，但他不仅可以通过 key 搜索，还可以用 value 来搜索。

```
#include <boost/bimap.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap persons;

  persons.insert(bimap::value_type("Boris", 31));
  persons.insert(bimap::value_type("Anton", 31));
  persons.insert(bimap::value_type("Caesar", 25));

  std::cout << persons.left.count("Boris") << std::endl;
  std::cout << persons.right.count(31) << std::endl;
}
```

- 下载源代码

在 `boost/bimap.hpp` 中定义的 `boost::bimap` 为我们提供了两个属性： `left` 和 `right` 来访问在 `boost::bimap` 统一的两个 `std::map` 类型的容器。在例子中， `left` 用 `std::string` 类型的 key 来访问容器，而 `right` 用到了 `int` 类型的 key。

除了支持用 left 和 right 对容器中的记录进行单独的访问， `boost::bimap` 还允许像下面的例子一样展示记录间的关联关系。

```
#include <boost/bimap.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap persons;

  persons.insert(bimap::value_type("Boris", 31));
  persons.insert(bimap::value_type("Anton", 31));
  persons.insert(bimap::value_type("Caesar", 25));

  for (bimap::iterator it = persons.begin(); it != persons.end(); -
    std::cout << it->left << " is " << it->right << " years old." <
}
```

- 下载源代码

对一个记录访问时， `left` 和 `right` 并不是必须的。 你也可以使用迭代器来访问每个记录中的 left 和 right 容器。

`std::map` 和 `std::multimap` 组合让你觉得似乎可以存储多个具有相同 key 值的记录， 但 `boost::bimap` 并没有这样做。 但这并不代表在 `boost::bimap` 存储两个具有相同 key 值的记录是不可能的。 严格来说， 那两个模板参数并不会对 `left` 和 `right` 的容器类型做出具体的规定。 如果像例子中那样并没有指定容器类型时， `boost::bimaps::set_of` 类型会缺省的使用。 跟 `std::map` 一样， 它要求记录有唯一的 key 值。

第一个 `boost::bimap` 例子也可以像下面这样写。

```
#include <boost/bimap.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>, boost::t
  bimap persons;

  persons.insert(bimap::value_type("Boris", 31));
  persons.insert(bimap::value_type("Anton", 31));
  persons.insert(bimap::value_type("Caesar", 25));

  std::cout << persons.left.count("Boris") << std::endl;
  std::cout << persons.right.count(31) << std::endl;
}
```

- 下载源代码

除了 `boost::bimaps::set_of` ， 你还可以用一些其他的容器类型来定制你的 `boost::bimap` 。

```
#include <boost/bimap.hpp>
#include <boost/bimap/multiset_of.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>, boost::b
  bimap persons;

  persons.insert(bimap::value_type("Boris", 31));
  persons.insert(bimap::value_type("Anton", 31));
  persons.insert(bimap::value_type("Caesar", 25));

  std::cout << persons.left.count("Boris") << std::endl;
  std::cout << persons.right.count(31) << std::endl;
}
```

- 下载源代码

代码中的容器使用了定义在 `boost/bimap/multiset_of.hpp` 中的 `boost::bimaps::multiset_of` 。 这个容器的操作和 `boost::bimaps::set_of` 差不了多少， 只是它不再要求 key 值是唯一的。 因此， 上面的例子将会在计算 age 为 31 的 person 数时输出： `2` 。

既然 `boost::bimaps::set_of` 会在定义 `boost::bimap` 被缺省的使用， 你没必要再显示的包含头文件： `boost/bimap/set_of.hpp` 。 但在使用其它类型的容器时， 你就必须要显示的包含一些相应的头文件了。

Boost.Bimap 还提供了类： `boost::bimaps::unordered_set_of` ， `boost::bimaps::unordered_multiset_of` ， `boost::bimaps::list_of` ， `boost::bimaps::vector_of` 和 `boost::bimaps::unconstrainted_set_of` 以供使用。 除了 `boost::bimaps::unconstrainted_set_of` ， 剩下的所有容器类型的使用方法都和他们在 C++ 标准里的版本一样。

```
#include <boost/bimap.hpp>
#include <boost/bimap/unconstrained_set_of.hpp>
#include <boost/bimap/support/lambda.hpp>
#include <iostream>
#include <string>

int main()
{
  typedef boost::bimap<std::string, boost::bimaps::unconstrained_se
  bimap persons;

  persons.insert(bimap::value_type("Boris", 31));
  persons.insert(bimap::value_type("Anton", 31));
  persons.insert(bimap::value_type("Caesar", 25));

  bimap::left_map::iterator it = persons.left.find("Boris");
  persons.left.modify_key(it, boost::bimaps::_key = "Doris");

  std::cout << it->first << std::endl;
}
```

- 下载源代码

`boost::bimaps::unconstrainted_set_of` 可以使 `boost::bimap` 的 `right` （也就是 age）值无法用来查找 person。 在这种特定的情况下， `boost::bimap` 可以被视为是一个 `std::map` 类型的容器。

虽然如此， 例子还是向我们展示了 `boost::bimap` 对于 `std::map` 的优越性。 因为 Boost.Bimap 是基于 Boost.MultiIndex 的， 你当然可以使用 Boost.MultiIndex 提供的所有函数。 例子中就用 `modify_key()` 修改了 key 值， 这在 `std::map` 中是不可能的。

请注意修改 key 值的以下细节： key 通过 `boost::bimaps::_key` 函数赋予了新值， 而 `boost::bimaps::_key` 是一个定义在 `boost/bimap/support/lambda.hpp` 中的 lambda 函数。 有关 lambda 函数， 详见：第 3 章 函数对象。

`boost/bimap/support/lambda.hpp` 还定义了 `boost::bimaps::_data` 。 函数 `modify_data()` 可以用来修改 `boost::bimap` 中的 value 值。

## 13.6. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 创建你的应用： 它支持指派员工到一家公司不同的部门。 存入一些示例记录， 然后通过指定员工来得到他所在的部门， 再通过指定部门来得到该部门的员工数。

2. 扩展你的应用：加入员工的 ID 号。ID 号必须是唯一的，保证在员工同名的情况下依然可以唯一的标识员工。通过指定 ID 号来得到某个员工的信息并把它输出以验证正确性。

# 第 14 章 数据结构

目录






## 14.1. 概述

在 Boost C++ 库中， 把一些类型定义为container显得不太合适， 所以就并没有放在 第 13 章 容器 里。 而把他们放在本章就比较合适了。 举例来说， `boost::tuple` 就扩展了 C++ 的数据类型 `std::pair` 用以储存多个而不只是两个值。

除了 `boost::tuple` ， 这一章还涵盖了类 `boost::any` 和 `boost::variant` 以储存那些不确定类型的值。 其中 `boost::any` 类型的变量使用起来就像弱类型语言中的变量一样灵活。 另一方面， `boost::variant` 类型的变量可以储存一些预定义的数据类型， 就像我们用 `union` 时候一样。

## 14.2. 元组

Boost.Tuple 库提供了一个更一般的版本的 `std::pair` —— `boost::tuple` 。 不过 `std::pair` 只能储存两个值而已， `boost::tuple` 则给了我们更多的选择。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string> person;
  person p("Boris", "Schaeling");
  std::cout << p << std::endl;
}
```

- 下载源代码

为了使用 `boost::tuple` ， 你必须要包含头文件： `boost/tuple/tuple.hpp` 。 若想要让元组和流一起使用， 你还需要包含头文件： `boost/tuple/tuple_io.hpp` 才行。

其实， `boost::tuple` 的用法基本上和 `std::pair` 一样。 就像我们在上面的例子里看到的那样， 两个值类型的 `std::string` 通过两个相应的模板参数存储在了元组里。

当然 `person` 类型也可以用 `std::pair` 来实现。 所有 `boost::tuple` 类型的对象都可以被写入流里。 再次强调， 为了使用流操作和各种流操作运算符， 你必须要包含头文件： `boost/tuple/tuple_io.hpp` 。 显然，我们的例子会输出： `(Boris Schaeling)` 。

`boost::tuple` 和 `std::pair` 之间最重要的一点不同点： 元组可以存储无限多个值！

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string, int> person;
  person p("Boris", "Schaeling", 43);
  std::cout << p << std::endl;
}
```

- 下载源代码

我们修改了实例， 现在的元组里不仅储存了一个人的firstname和lastname， 还加上了他的鞋子的尺码。 现在， 我们的例子将会输出： `(Boris Schaeling 43)` 。

就像 `std::pair` 有辅助函数 `std::make_pair()` 一样， 一个元组也可以用它的辅助函数 `boost::make_tuple()` 来创建。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <iostream>

int main()
{
  std::cout << boost::make_tuple("Boris", "Schaeling", 43) << std::
}
```

- 下载源代码

就像下面的例子所演示的那样， 一个元组也可以存储引用类型的值。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boris";
  std::cout << boost::make_tuple(boost::ref(s), "Schaeling", 43) <<
}
```

- 下载源代码

因为 "Schaeling" 和 43 是按值传递的，所以就直接存储在了元组中。 与他们不同的是： person 的第一个元素是一个指向 `s` 的引用。 Boost.Ref 中的 `boost::ref()` 就是用来创建这样的引用的。 相对的， 要创建一个常量的引用的时候， 你需要使用 `boost::cref()` 。

在学习了创建元组的方法之后， 让我们来了解一下访问元组中元素的方式。 `std::pair` 只包含两个元素， 故可以使用属性 `first` 和 `second` 来访问其中的元素。 但元组可以包含无限多个元素， 显然， 我们需要用另一种方式来解决访问的问题。

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string, int> person;
  person p = boost::make_tuple("Boris", "Schaeling", 43);
  std::cout << p.get<0>() << std::endl;
  std::cout << boost::get<0>(p) << std::endl;
}
```

- 下载源代码

我们可以用两种方式来访问元组中的元素： 使用成员函数 `get()` ， 或者将元组传给一个独立的函数 `boost::get()` 。 使用这两种方式时， 元素的索引值都是通过模板参数来指定的。 例子中就分别使用了这两种方式来访问 `p` 中的第一个元素。 因此， `Boris` 会被输出两次。

另外， 对于索引值合法性的检查会在编译期执行， 故访问非法的索引值会引起编译期错误而不是运行时的错误。

对于元组中元素的修改，你同样可以使用 `get()` 和 `boost::get()` 函数。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string, int> person;
  person p = boost::make_tuple("Boris", "Schaeling", 43);
  p.get<1>() = "Becker";
  std::cout << p << std::endl;
}
```

- 下载源代码

`get()` 和 `boost::get()` 都会返回一个引用值。例子中修改了 lastname 之后将会输出： `(Boris Becker 43)` 。

Boost.Tuple 除了重载了流操作运算符以外，还为我们提供了比较运算符。为了使用它们，你必须要包含相应的头文件： `boost/tuple/tuple_comparison.hpp` 。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, std::string, int> person;
  person p1 = boost::make_tuple("Boris", "Schaeling", 43);
  person p2 = boost::make_tuple("Boris", "Becker", 43);
  std::cout << (p1 != p2) << std::endl;
}
```

- 下载源代码

上面的例子将会输出 `1` 因为两个元组 `p1` 和 `p2` 是不同的。

同时，头文件 `boost/tuple/tuple_comparison.hpp` 还定义了一些其他的比较操作，比如用来做字典序比较的大于操作等。

Boost.Tuple 还提供了一种叫做 Tier 的特殊元组。Tier 的特殊之处在于它包含的所有元素都是引用类型的。它可以通过构造函数 `boost::tie()` 来创建。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, std::string&, int&> person;

  std::string firstname = "Boris";
  std::string surname = "Schaeling";
  int shoesize = 43;
  person p = boost::tie(firstname, surname, shoesize);
  surname = "Becker";
  std::cout << p << std::endl;
}
```

- 下载源代码

上面的例子创建了一个 tier `p` ，他包含了三个分别指向 `firstname` ，`surname` 和 `shoesize` 的引用值。在修改变量 `surname` 的同时， tier 也会跟着改变。

就像下面的例子展示的那样，你当然可以用 `boost::make_tuple()` 和 `boost::ref()` 来代替构造函数 `boost::tie()` 。

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, std::string&, int&> person;

  std::string firstname = "Boris";
  std::string surname = "Schaeling";
  int shoesize = 43;
  person p = boost::make_tuple(boost::ref(firstname), boost::ref(su
  surname = "Becker";
  std::cout << p << std::endl;
}
```

- 下载源代码

`boost::tie()` 在一定程度上简化了语法， 同时， 也可以用作“拆箱”元组。 在接下来的这个例子里， 元组中的各个元素就被很方便的“拆箱”并直接赋给了其他变量。

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

boost::tuple<std::string, int> func()
{
  return boost::make_tuple("Error message", 2009);
}

int main()
{
  std::string errmsg;
  int errcode;

  boost::tie(errmsg, errcode) = func();
  std::cout << errmsg << ": " << errcode << std::endl;
}
```

- 下载源代码

通过使用 `boost::tie()` ， 元组中的元素：字符串“Error massage”和错误代码“2009”就很方便地经 `func()` 的返回值直接赋给了 `errmsg` 和 `errcode` 。

## 14.3. Boost.Any

像 C++ 这样的强类型语言要求给每个变量一个确定的类型。 而以 JavaScript 为代表的弱类型语言却不这样做， 弱类型的每个变量都可以存储数组、 布尔值、 或者是字符串。

库 Boost.Any 给我们提供了 `boost::any` 类， 让我们可以在 C++ 中像 JavaScript 一样的使用弱类型的变量。

```
#include <boost/any.hpp>

int main()
{
  boost::any a = 1;
  a = 3.14;
  a = true;
}
```

- 下载源代码

为了使用 `boost::any` ，你必须要包含头文件： `boost/any.hpp` 。接下来，你就可以定义和使用 `boost::any` 的对象了。

需要注明的是： `boost::any` 并不能真的存储任意类型的值； Boost.Any 需要一些特定的前提条件才能工作。 任何想要存储在 `boost::any` 中的值，都必须是可拷贝构造的。 因此，想要在 `boost::any` 存储一个字符串类型的值， 就必须要用到 `std::string` ， 就像在下面那个例子中做的一样。

```
#include <boost/any.hpp>
#include <string>

int main()
{
  boost::any a = 1;
  a = 3.14;
  a = true;
  a = std::string("Hello, world!");
}
```

- 下载源代码

如果你企图把字符串 "Hello, world!" 直接赋给 `a` ， 你的编译器就会报错， 因为由基类型 `char` 构成的字符串在 C++ 中并不是可拷贝构造的。

想要访问 `boost::any` 中具体的内容， 你必须要使用转型操作： `boost::any_cast` 。

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  boost::any a = 1;
  std::cout << boost::any_cast<int>(a) << std::endl;
  a = 3.14;
  std::cout << boost::any_cast<double>(a) << std::endl;
  a = true;
  std::cout << boost::any_cast<bool>(a) << std::endl;
}
```

- 下载源代码

通过由模板参数传入 `boost::any_cast` 的值， 变量会被转化成相应的类型。 一旦你指定了一种非法的类型， 该操作会抛出 `boost::bad_any_cast` 类型的异常。

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::any a = 1;
    std::cout << boost::any_cast<float>(a) << std::endl;
  }
  catch (boost::bad_any_cast &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

上面的例子就抛出了一个异常，因为 `float` 并不能匹配原本存储在 `a` 中的 `int` 类型。记住，在任何情况下都保证 `boost::any` 中的类型匹配是很重要的。在没有通过模板参数指定 `short` 或 `long` 类型时，同样会有异常抛出。

既然 `boost::bad_any_cast` 继承自 `std::bad_cast`，`catch` 当然也可以捕获相应类型的异常。

想要检查 `boost::any` 是否为空，你可以使用 `empty()` 函数。想要确定其中具体的类型信息，你可以使用 `type()` 函数。

```
#include <boost/any.hpp>
#include <typeinfo>
#include <iostream>

int main()
{
  boost::any a = 1;
  if (!a.empty())
  {
    const std::type_info &ti = a.type();
    std::cout << ti.name() << std::endl;
  }
}
```

- 下载源代码

上面的例子同时用到了 `empty()` 和 `type()` 函数。`empty()` 将会返回一个布尔值，而 `type()` 则会返回一个在 `typeinfo` 中定义的 `std::type_info` 值。

作为对这一节的总结， 最后一个例子会向你展示怎样用 `boost::any_cast` 来定义一个指向 `boost::any` 中内容的指针。

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  boost::any a = 1;
  int *i = boost::any_cast<int>(&a);
  std::cout << *i << std::endl;
}
```

- 下载源代码

你需要做的就是传递一个 `boost::any` 类型的指针， 作为 `boost::any_cast` 的参数； 模板参数却没有任何改动。

## 14.4. Boost.Variant

Boost.Variant 和 Boost.Any 之间的不同点在于 Boost.Any 可以被视为任意的类型， 而 Boost.Variant 只能被视为固定数量的类型。 让我们来看下面这个例子。

```
#include <boost/variant.hpp>

int main()
{
  boost::variant<double, char> v;
  v = 3.14;
  v = 'A';
}
```

- 下载源代码

Boost.Variant 为我们提供了一个定义在 `boost/variant.hpp` 中的类： `boost::variant` 。 既然 `boost::variant` 是一个模板， 你必须要指定至少一个参数。 Variant 所存储的数据类型就由这些参数来指定。 上面的例子就给 `v` 指定了 `double` 类型和 `char` 类型。 注意， 一旦你将一个 `int` 值赋给了 `v` ， 你的代码将不会编译通过。

当然， 上面的例子也可以用一个 `union` 类型来实现， 但是与 union 不同的是： `boost::variant` 可以储存像 `std::string` 这样的 class 类型的数据。

```
#include <boost/variant.hpp>
#include <string>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  v = 'A';
  v = "Hello, world!";
}
```

- 下载源代码

要访问 `v` 中的数据， 你可以使用独立的 `boost::get()` 函数。

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << boost::get<double>(v) << std::endl;
  v = 'A';
  std::cout << boost::get<char>(v) << std::endl;
  v = "Hello, world!";
  std::cout << boost::get<std::string>(v) << std::endl;
}
```

- 下载源代码

`boost::get()` 需要传入一个模板参数来指明你需要返回的数据类型。 若是指定了一个非法的类型， 你会遇到一个运行时而不是编译期的错误。

所有 `boost::variant` 类型的值都可以被直接写入标准输入流这样的流中， 这可以在一定程度上让你避开运行时错误的风险。

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << v << std::endl;
  v = 'A';
  std::cout << v << std::endl;
  v = "Hello, world!";
  std::cout << v << std::endl;
}
```

- 下载源代码

想要分别处理各种不同类型的数据， Boost.Variant 为我们提供了一个名为 `boost::apply_visitor()` 的函数。

```
#include <boost/variant.hpp>
#include <boost/any.hpp>
#include <vector>
#include <string>
#include <iostream>

std::vector<boost::any> vector;

struct output :
  public boost::static_visitor<>
{
  void operator()(double &d) const
  {
    vector.push_back(d);
  }

  void operator()(char &c) const
  {
    vector.push_back(c);
  }

  void operator()(std::string &s) const
  {
    vector.push_back(s);
  }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output(), v);
  v = 'A';
  boost::apply_visitor(output(), v);
  v = "Hello, world!";
  boost::apply_visitor(output(), v);
}
```

- 下载源代码

`boost::apply_visitor()` 第一个参数需要传入一个继承自 `boost::static_visitor` 类型的对象。 这个类必须要重载 `operator()()` 运算符来处理 `boost::variant` 每个可能的类型。 相应的， 例子中的 `v` 就重载了三次 operator() 来处理三种可能的类型： `double` ， `char` 和 `std::string` 。

再仔细看代码， 不难发现 `boost::static_visitor` 是一个模板。 那么， 当 `operator()()` 有返回值的时候， 就必须返回一个模板才行。 如果 operator() 像例子那样没有返回值时， 你就不需要模板了。

`boost::apply_visitor()` 的第二个参数是一个 `boost::variant` 类型的值。

在使用时， `boost::apply_visitor()` 会自动调用跟第二个参数匹配的 `operator()()` 。 示例程序中的 `boost::apply_visitor()` 就自动调用了三个不同的 operator 第一个是 `double` 类型的， 第二个是 `char` 最后一个是 `std::string` 。

`boost::apply_visitor()` 的优点不只是“自动调用匹配的函数”这一点。 更有用的是， `boost::apply_visitor()` 会确认是否 `boost::variant` 中的每个可能值都定义了相应的函数。 如果你忘记重载了任何一个函数， 代码都不会编译通过。

当然， 如果对每种类型的操作都是一样的， 你也可以像下面的示例一样使用一个模板来简化你的代码。

```
#include <boost/variant.hpp>
#include <boost/any.hpp>
#include <vector>
#include <string>
#include <iostream>

std::vector<boost::any> vector;

struct output :
  public boost::static_visitor<>
{
  template <typename T>
  void operator()(T &t) const
  {
    vector.push_back(t);
  }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output(), v);
  v = 'A';
  boost::apply_visitor(output(), v);
  v = "Hello, world!";
  boost::apply_visitor(output(), v);
}
```

- 下载源代码

既然 `boost::apply_visitor()` 可以在编译期确定代码的正确性， 你就该更多的使用它而不是 `boost::get()` 。

## 14.5. 练习

You can buy solutions to all exercises in this book as a ZIP file.

1. 自定义一种数据类型： `configuration` 它可以存储一个 name-value 对。Name 为 `std::string` 类型，而 value 可为 `std::string` 或者 `int` 或者 `float` 类型。在 `main()` 函数里，用 `configuration` 存储下列 name-value 对: path=C:\Windows, version=3， pi=3.1415。通过向便准输出流输出来验证你对数据类型的设计。

2. 在输出后，将对象中的 path 修改为 C:\Windows\System。再次向标准输出流输出以验证你的设计。

# 第 15 章 错误处理

## 目录





## 15.1. 概述

在执行时会有潜在失败可能的每个函数都需要一种合适的方式和它的调用者进行交互。 在C++中，这一步是通过返回值或抛出一个异常来完成的。 作为常识，返回值经常用在处理非错误的异常中。 调用者通过返回值作出相应的反馈。

异常被通常用来标示出未预期的异常情况。 一个很好的例子是在错误的使用 `new` 时将抛出的一个动态内存分配异常类型 `std::bad_alloc` 。 由于内存的分配通常不会出现任何问题，如果总是检查返回值将会变得异常累赘。

本章介绍了两种可以帮助开发者利用错误处理的Boost C++库：其中 Boost.System 可以由特定操作系统平台的错误代码转换出跨平台的错误代码。 借助于 Boost.System，函数基于某个特定操作系统的返回值类型可以被转换成为跨平台的类型。 另外，Boost.Exception 允许给任何异常添加额外的信息，以便利用 `catch` 相应的处理程序更好的对异常作出反应。

## 15.2. Boost.System

Boost.System 是一个定义了四个类的小型库，用以识别错误。 `boost::system::error_code` 是一个最基本的类，用于代表某个特定操作系统的异常。 由于操作系统通常枚举异常， `boost::system::error_code` 中以变量的形式保存错误代码 `int` 。 下面的例子说明了如何通过访问 Boost.Asio 类来使用这个类。

```
#include <boost/system/error_code.hpp>
#include <boost/asio.hpp>
#include <iostream>
#include <string>

int main()
{
  boost::system::error_code ec;
  std::string hostname = boost::asio::ip::host_name(ec);
  std::cout << ec.value() << std::endl;
}
```

- 下载源代码

Boost.Asio 提供了独立的函数 `boost::asio::ip::host_name()` 可以返回正在执行的应用程序名。

`boost::system::error_code` 类型的一个对象可以作为单独的参数传递给 `boost::asio::ip::host_name()` 。 如果当前的操作系统函数失败， 这个参数包含相关的错误代码。 也可以通过调用 `boost::asio::ip::host_name()` 而不使用任何参数，以防止错误代码是非相关的。

事实上在Boost 1.36.0中 `boost::asio::ip::host_name()` 是有问题的，然而它可以当作一个很好的例子。 即使当前操作系统函数成功返回了计算机名，这个函数它也可能返回一个错误代码。 由于在Boost 1.37.0中解决了这个问题，现在可以放心使用 `boost::asio::ip::host_name()` 了。

由于错误代码仅仅是一个数值，因此可以借助于 `value()` 方法得到它。 由于错误代码0通常意味着没有错误，其他的值的意义则依赖于操作系统并且需要查看相关手册。

如果使用Boost 1.36.0， 并且用Visual Studio 2008在Windows XP环境下编译以上应用程序将不断产生错误代码14（没有足够的存储空间以完成操作） 。 即使函数 `boost::asio::ip::host_name()` 成功决定了计算机名，也会报出错误代码14。 事实上这是因为函数 `boost::asio::ip::host_name()` 的实现有问题。

除了 `value()` 方法之外, 类 `boost::system::error_code` 提供了方法 `category()` 。 这个方法可返回一个在 Boost.System 中定义的二级对象: `boost::system::category` 。

错误代码是简单的数值。 操作系统开发商，例如微软，可以保证系统错误代码的特异性。 对于任何开发商来说，在所有现有应用程序中保持错误代码的独一无二是几乎不可能的。 他需要一个包含有所有软件开发者的错误代码中心数据库，以防止在不同的方案下重复使用相同的代码。 当然这是不实际的。 这是错误分类表存在的缘由。

类型 `boost::system::error_code` 的错误代码总是属于可以使用 `category()` 方法获取的分类。 通过预定义的对象 `boost::system::system_category` 来表示操作系统的错误。

通过调用 `category()` 方法，可以返回预定义变量 `boost::system::system_category` 的一个引用。 它允许获取关于分类的特定信息。 例如在使用的是 system 分类的情况下，通过使用 `name()` 方法将得到它的名字 `system` 。

```
#include <boost/system/error_code.hpp>
#include <boost/asio.hpp>
#include <iostream>
#include <string>

int main()
{
  boost::system::error_code ec;
  std::string hostname = boost::asio::ip::host_name(ec);
  std::cout << ec.value() << std::endl;
  std::cout << ec.category().name() << std::endl;
}
```

- 下载源代码

通过错误代码和错误分类识别出的错误是独一无二的。 由于仅仅在错误分类中的错误代码是必须唯一的，程序员应当在希望定义某个特定应用程序的错误代码时创建一个新的分类。 这使得任何错误代码都不会影响到其他开发者的错误代码。

```
#include <boost/system/error_code.hpp>
#include <iostream>
#include <string>

class application_category :
  public boost::system::error_category
{
public:
  const char *name() const { return "application"; }
  std::string message(int ev) const { return "error message"; }
};

application_category cat;

int main()
{
  boost::system::error_code ec(14, cat);
  std::cout << ec.value() << std::endl;
  std::cout << ec.category().name() << std::endl;
}
```

- 下载源代码

通过创建一个派生于 `boost::system::error_category` 的类以及实现作为新分类的所necessary的接口的不同方法可以定义一个新的错误分类。 由于方法 `name()` 和 `message()` 在类 `boost::system::error_category` 中被定义为纯虚拟函数，所以它们是必须提供的。 至于额外的方法，在必要的条件下，可以重载相对应的默认行为。

当方法 `name()` 返回错误分类名时，可以使用方法 `message()` 来获取针对某个错误代码的描述。 不像之前的那个例子，参数 `ev` 往往被用于返回基于错误代码的描述。

新创建的错误分类的对象可以被用来初始化相应的错误代码。 本例中定义了一个用于新分类 `application_category` 的错误代码 `ec` 。 然而错误代码14不再是系统错误；他的意义被开发者指定为新的错误分类。

`boost::system::error_code` 包含了一个叫作 `default_error_condition()` 的方法，它可以返回 `boost::system::error_condition` 类型的对象。 `boost::system::error_condition` 的接口几乎与 `boost::system::error_code` 相同。 唯一的差别是只有类 `boost::system::error_code` 提供了方法 `default_error_condition()` 。

```
#include <boost/system/error_code.hpp>
#include <boost/asio.hpp>
#include <iostream>
#include <string>

int main()
{
  boost::system::error_code ec;
  std::string hostname = boost::asio::ip::host_name(ec);
  boost::system::error_condition ecnd = ec.default_error_condition(
  std::cout << ecnd.value() << std::endl;
  std::cout << ecnd.category().name() << std::endl;
}
```

- 下载源代码

`boost::system::error_condition` 的使用方法与 `boost::system::error_code` 类似。 对象 `boost::system::error_condition` 的 `value()` 和 `category()` 方法都可以像上面的例子中那样调用。

有或多或少两个相同的类的原因很简单：当类 `boost::system::error_code` 被当作当前平台的错误代码时， 类 `boost::system::error_condition` 可以被用作获取跨平台的错误代码。 通过调用 `default_error_condition()` 方法，可以把依赖于某个平台的的错误代码转换成 `boost::system::error_condition` 类型的跨平台的错误代码。

如果执行以上应用程序，它将显示数字12以及错误分类 `GENERIC` 。 依赖于平台的错误代码14被转换成了跨平台的错误代码12。 借助于 `boost::system::error_condition` ，可以总是使用相同的数字表示错误，无视当前操作系统。 当Windows报出错误14时，其他操作系统可能会对相同的错误报出错误代码25。 使用 `boost::system::error_condition` ，总是对这个错误报出错误代码12。

最后 Boost.System 提供了类 `boost::system::system_error` ，它派生于 `std::runtime_error` 。 它可被用来传送发生在异常里类型为 `boost::system::error_code` 的错误代码。

```
#include <boost/asio.hpp>
#include <boost/system/system_error.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::asio::ip::host_name() << std::endl;
  }
  catch (boost::system::system_error &e)
  {
    boost::system::error_code ec = e.code();
    std::cerr << ec.value() << std::endl;
    std::cerr << ec.category().name() << std::endl;
  }
}
```

- 下载源代码

独立的函数 `boost::asio::ip::host_name()` 是以两种方式提供的：一种是需要类型为 `boost::system::error_code` 的参数，另一种不需要参数。 第二个版本将在错误发生时抛出 `boost::system::system_error` 类型的异常。 异常传出类型为 `boost::system::error_code` 的相应错误代码。

## 15.3. Boost.Exception

Boost.Exception 库提供了一个新的异常类 `boost::exception` 允许给一个抛出的异常添加信息。 它被定义在文件 `boost/exception/exception.hpp` 中。 由于 Boost.Exception 中的类和函数分布在不同的头文件中， 下面的例子中将使用 `boost/exception/all.hpp` 以避免一个一个添加头文件。

```
#include <boost/exception/all.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/shared_array.hpp>
#include <exception>
```

```
#include <string>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_in

class allocation_failed :
  public boost::exception,
  public std::exception
{
public:
  allocation_failed(std::size_t size)
    : what_("allocation of " + boost::lexical_cast<std::string>(siz
  {
  }

  virtual const char *what() const throw()
  {
    return what_.c_str();
  }

private:
  std::string what_;
};

boost::shared_array<char> allocate(std::size_t size)
{
  if (size > 1000)
    throw allocation_failed(size);
  return boost::shared_array<char>(new char[size]);
}

void save_configuration_data()
{
  try
  {
    boost::shared_array<char> a = allocate(2000);
    // saving configuration data ...
  }
  catch (boost::exception &e)
  {
    e << errmsg_info("saving configuration data failed");
    throw;
  }
}

int main()
{
  try
  {
    save_configuration_data();
  }
  catch (boost::exception &e)
  {
```

```
        std::cerr << boost::diagnostic_information(e);
    }
}
```

- 下载源代码

这个例子在 `main()` 中调用了一个函数 `save_configuration_data()` ，它调回了 `allocate()` 。 `allocate()` 函数动态分配内存，而它检查是否超过某个限度。 这个限度在本例中被设定为1,000个字节。

如果 `allocate()` 被调用的值大于1,000，将会抛出 `save_configuration_data()` 函数里的相应异常。 正如注释中所标识的那样，这个函数把配置数据被存储在动态分配的内存中。

事实上，这个例子的目的是通过抛出异常以示范 Boost.Exception。 这个通过 `allocate()` 抛出的异常是 `allocation_failed` 类型的，而且它同时继承了 `boost::exception` 和 `std::exception` 。

当然，也不是一定要派生于 `std::exception` 异常的。 为了把它嵌入到现有的框架中，异常 `allocation_failed` 可以派生于其他类的层次结构。 当通过C++标准来定义以上例子的类层次结构的时候， 单独从 `boost::exception` 中派生出 `allocation_failed` 就足够了。

当抛出 `allocation_failed` 类型的异常的时候，分配内存的大小是存储在异常中的，以缓解相应应用程序的调试。 如果想通过 `allocate()` 分配获取更多的内存空间，那么可以很容易发现导致异常的根本原因。

如果仅仅通过一个函数(例子中的函数 `save_configuration_data()` )来调用 `allocate()` ，这个信息足以找到问题的所在。 然而，在有许多函数调用 `allocate()` 以动态分配内存的更加复杂的应用程序中，这个信息不足以高效的调试应用程序。 在这些情况下，它最好能有助于找到哪个函数试图分配 `allocate()` 所能提供空间之外的内存。 向异常中添加更多的信息，在这些情况下，将非常有助于进程的调试。

有挑战性的是，函数 `allocate()` 中并没有调用者名等信息，以把它加入到相关的异常中。

Boost.Exception 提供了如下的解决方案：对于任何一个可以添加到异常中的信息，可以通过定义一个派生于 `boost::error_info` 的数据类型，来随时向这个异常添加信息。

`boost::error_info` 是一个需要两个参数的模板，第一个参数叫做标签(tag)，特定用来识别新建的数据类型。 通常是一个有特定名字的结构体。 第二个参数是与存储于异常中的数据类型信息相关的。

这个应用程序定义了一个新的数据类型 `errmsg_info` ，可以通过 `tag_errmsg` 结构来特异性的识别，它存储着一个 `std::string` 类型的字符串。

在 save_configuration_data() 的 catch 句柄中，通过获取 tag_errmsg 以创建一个对象，它通过字符串 "saving configuration data failed" 进行初始化，以便通过 operator<<() 操作符向异常 boost::exception 中加入更多信息。 然后这个异常被相应的重新抛出。

现在，这个异常不仅包含有需要动态分配的内存大小，而且对于错误的描述被填入到 save_configuration_data() 函数中。 在调试时，这个描述显然很有帮助，因为可以很容易明白哪个函数试图分配更多的内存。

为了从一个异常中获取所有可用信息，可以像例子中那样在 main() 的 catch 句柄中使用函数 boost::diagnostic_information() 。 对于每个异常，函数 boost::diagnostic_information() 不仅调用 what() 而且获取所有附加信息存储到异常中。 返回一个可以在标准输出中写入的 std::string 字符串。

以上程序通过Visual C++ 2008编译会显示如下的信息：

```
Throw in function (unknown)
Dynamic exception type: class allocation_failed
std::exception::what: allocation of 2000 bytes failed
[struct tag_errmsg *] = saving configuration data failed
```

正如我们所看见的，数据包含了异常的数据类型，通过 what() 方法获取到错误信息，以及包括相应结构体名的描述。

boost::diagnostic_information() 函数在运行时检查一个给定的异常是否派生于 std::exception 。 只会在派生于 std::exception 的条件下调用 what() 方法。

抛出异常类型 allocation_failed 的函数名会被指定为"unknown"(未知)信息。

Boost.Exception 提供了一个用以抛出异常的宏，它包含了函数名，以及如文件名、行数的附加信息。

```
#include <boost/exception/all.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/shared_array.hpp>
#include <exception>
#include <string>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_i

class allocation_failed :
  public std::exception
{
public:
  allocation_failed(std::size_t size)
    : what_("allocation of " + boost::lexical_cast<std::string>(si
  {
  }
```

```
  virtual const char *what() const throw()
  {
    return what_.c_str();
  }

private:
  std::string what_;
};

boost::shared_array<char> allocate(std::size_t size)
{
  if (size > 1000)
    BOOST_THROW_EXCEPTION(allocation_failed(size));
  return boost::shared_array<char>(new char[size]);
}

void save_configuration_data()
{
  try
  {
    boost::shared_array<char> a = allocate(2000);
    // saving configuration data ...
  }
  catch (boost::exception &e)
  {
    e << errmsg_info("saving configuration data failed");
    throw;
  }
}

int main()
{
  try
  {
    save_configuration_data();
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

- 下载源代码

通过使用宏 `BOOST_THROW_EXCEPTION` 替代 `throw`， 如函数名、文件名、行数之类的附加信息将自动被添加到异常中。但这仅仅在编译器支持宏的情况下有效。当通过C++标准定义 `__FILE__` 和 `__LINE__` 之类的宏时，没有用于返回当前

函数名的标准化的宏。 由于许多编译器制造商提供这样的宏， `BOOST_THROW_EXCEPTION` 试图识别当前编译器，从而利用相对应的宏。 使用 Visual C++ 2008 编译时，以上应用程序显示以下信息：

```
.\main.cpp(31): Throw in function class boost::shared_array<char> _
Dynamic exception type: class boost::exception_detail::clone_impl<s
std::exception::what: allocation of 2000 bytes failed
[struct tag_errmsg *] = saving configuration data failed
```

即使 `allocation_failed` 类不再派生于 `boost::exception` 代码的编译也不会产生错误。 `BOOST_THROW_EXCEPTION` 获取到一个能够动态识别是否派生于 `boost::exception` 的函数 `boost::enable_error_info()` 。 如果不是，他将自动建立一个派生于特定类和 `boost::exception` 的新异常类型。 这个机制使得以上信息中不仅仅显示内存分配异常 `allocation_failed` 。

最后，这个部分包含了一个例子，它选择性的获取了添加到异常中的信息。

```
#include <boost/exception/all.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/shared_array.hpp>
#include <exception>
#include <string>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_in

class allocation_failed :
  public std::exception
{
public:
  allocation_failed(std::size_t size)
    : what_("allocation of " + boost::lexical_cast<std::string>(siz
  {
  }

  virtual const char *what() const throw()
  {
    return what_.c_str();
  }

private:
  std::string what_;
};

boost::shared_array<char> allocate(std::size_t size)
{
  if (size > 1000)
    BOOST_THROW_EXCEPTION(allocation_failed(size));
  return boost::shared_array<char>(new char[size]);
```

```
}

void save_configuration_data()
{
  try
  {
    boost::shared_array<char> a = allocate(2000);
    // saving configuration data ...
  }
  catch (boost::exception &e)
  {
    e << errmsg_info("saving configuration data failed");
    throw;
  }
}

int main()
{
  try
  {
    save_configuration_data();
  }
  catch (boost::exception &e)
  {
    std::cerr << *boost::get_error_info<errmsg_info>(e);
  }
}
```

- 下载源代码

这个例子并没有使用函数 `boost::diagnostic_information()` 而是使用 `boost::get_error_info()` 函数来直接获取错误信息的类型 `errmsg_info` 。函数 `boost::get_error_info()` 用于返回 `boost::shared_ptr` 类型的智能指针。 如果传递的参数不是 `boost::exception` 类型的，返回的值将是相应的空指针。 如果 `BOOST_THROW_EXCEPTION` 宏总是被用来抛出异常，派生于 `boost::exception` 的异常是可以得到保障的——在这些情况下没有必要去检查返回的智能指针是否为空。

# 第 16 章 类型转换操作符

## 目录






## 16.1. 概述

C++标准定义了四种类型转换操作符: `static_cast` , `dynamic_cast` , `const_cast` 和 `reinterpret_cast` 。 Boost.Conversion 和 Boost.NumericConversion 这两个库特别为某些类型转换定义了额外的类型转换操作符。

## 16.2. Boost.Conversion

Boost.Conversion 库由两个文件组成。分别在 `boost/cast.hpp` 文件中定义了 `boost::polymorphic_cast` 和 `boost::polymorphic_downcast` 这两个类型转换操作符, 在 `boost/lexical_cast.hpp` 文件中定义了 `boost::lexical_cast` 。

`boost::polymorphic_cast` 和 `boost::polymorphic_downcast` 是为了使原来用 `dynamic_cast` 实现的类型转换更加具体。具体细节, 如下例所示。

```
struct father
{
  virtual ~father() { };
};

struct mother
{
  virtual ~mother() { };
};

struct child :
  public father,
  public mother
{
};

void func(father *f)
{
  child *c = dynamic_cast<child*>(f);
}

int main()
{
  child *c = new child;
  func(c);

  father *f = new child;
  mother *m = dynamic_cast<mother*>(f);
}
```

- 下载源代码

本例使用 `dynamic_cast` 类型转换操作符两次: 在 `func()` 函数中，它将指向父类的指针转换为指向子类的指针。在 `main()` 中, 它将一个指向父类的指针转为指向另一个父类的指针。第一个转换称为向下转换(downcast)，第二个转换称为交叉转换(cross cast)。

通过使用 Boost.Conversion 的类型转换操作符，可以将向下转换和交叉转换区分开来。

```
#include <boost/cast.hpp>

struct father
{
  virtual ~father() { };
};

struct mother
{
  virtual ~mother() { };
};

struct child :
  public father,
  public mother
{
};

void func(father *f)
{
  child *c = boost::polymorphic_downcast<child*>(f);
}

int main()
{
  child *c = new child;
  func(c);

  father *f = new child;
  mother *m = boost::polymorphic_cast<mother*>(f);
}
```

- 下载源代码

`boost::polymorphic_downcast` 类型转换操作符只能用于向下转换。 它内部使用 `static_cast` 实现类型转换。 由于 `static_cast` 并不动态检查类型转换是否合法，所以 `boost::polymorphic_downcast` 应该只在类型转换是安全的情况下使用。 在调试(debug builds)模式下， `boost::polymorphic_downcast` 实际上在 `assert ()` 函数中使用 `dynamic_cast` 验证类型转换是否合法。 请注意这种合法性检测只在定义了 `NDEBUG` 宏的情况下执行，这通常是在调试模式下。

向下转换最好使用 `boost::polymorphic_downcast` , 那么 `boost::polymorphic_cast` 就是交叉转换所需要的了。 由于 `dynamic_cast` 是唯一能实现交叉转换的类型转换操作符， `boost::polymorphic_cast` 内部使用了它。 由于 `boost::polymorphic_cast` 能够在错误的时候抛出 `std::bad_cast` 类型的异常，所以优先使用这个类型转换操作符还是很有必要的。相反， `dynamic_cast` 在类型转换失败使将返回0。 避免手工验证返回值， `boost::polymorphic_cast` 提供了自动化的替代方式。

`boost::polymorphic_downcast` 和 `boost::polymorphic_cast` 只在指针必须转换的时候使用；否则，必须使用 `dynamic_cast` 执行转换。 由于 `boost::polymorphic_downcast` 是基于 `static_cast` ，所以它不能够，比如说，将父类对象转换为子类对象。 如果转换的类型不是指针，则使用 `boost::polymorphic_cast` 执行类型转换也没有什么意义，而在这种情况下使用 `dynamic_cast` 还会抛出一个 `std::bad_cast` 异常。

虽然所有的类型转换都可用 `dynamic_cast` 实现，可 `boost::polymorphic_downcast` 和 `boost::polymorphic_cast` 也不是真正随意使用的。 Boost.Conversion 还提供了另外一种在实践中很有用的类型转换操作符。 体会一下下面的例子。

```
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = boost::lexical_cast<std::string>(169);
  std::cout << s << std::endl;
  double d = boost::lexical_cast<double>(s);
  std::cout << d << std::endl;
}
```

- 下载源代码

类型转换操作符 `boost::lexical_cast` 可将数字转换为其他类型。 例子首先将整数169转换为字符串，然后将字符串转换为浮点数。

`boost::lexical_cast` 内部使用流(streams)执行转换操作。 因此，只有那些重载了 `operator<<()` 和 `operator>>()` 这两个操作符的类型可以转换。 使用 `boost::lexical_cast` 的优点是类型转换出现在一行代码之内，无需手工操作流(streams)。 由于流的用法对于类型转换不能立刻理解代码含义，而 `boost::lexical_cast` 类型转换操作符还可以使代码更有意义，更加容易理解。

请注意 `boost::lexical_cast` 并不总是访问流(streams)；它自己也优化了一些数据类型的转换。

如果转换失败，则抛出 `boost::bad_lexical_cast` 类型的异常，它继承自 `std::bad_cast` 。

```
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

int main()
{
  try
  {
    int i = boost::lexical_cast<int>("abc");
    std::cout << i << std::endl;
  }
  catch (boost::bad_lexical_cast &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

本例由于字符串 "abc" 不能转换为 `int` 类型的数字而抛出异常。

## 16.3. Boost.NumericConversion

Boost.NumericConversion 可将一种数值类型转换为不同的数值类型。 在C++里，这种转换可以隐式地发生，如下面例所示。

```
#include <iostream>

int main()
{
  int i = 0x10000;
  short s = i;
  std::cout << s << std::endl;
}
```

- 下载源代码

由于从 `int` 到 `short` 的类型转换自动产生，所以本例编译没有错误。 虽然本例可以运行，但结果由于依赖具体的编译器实现而结果无法预期。 数字 `0x10000` 对于变量 `i` 来说太大而不能存储在 `short` 类型的变量中。 依据C++标准，这个操作的结果是实现定义的("implementation defined")。 用Visual C++ 2008编译，应用程序显示的是 `0` 。 `s` 的值当然不同于 `i` 的值。

为避免这种数值转换错误，可以使用 `boost::numeric_cast` 类型转换操作符。

```
#include <boost/numeric/conversion/cast.hpp>
#include <iostream>

int main()
{
  try
  {
    int i = 0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << std::endl;
  }
  catch (boost::numeric::bad_numeric_cast &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

`boost::numeric_cast` 的用法与C++类型转换操作符非常相似。 当然需要包含正确的头文件；就是 `boost/numeric/conversion/cast.hpp` 。

`boost::numeric_cast` 执行与C++相同的隐式转换操作。 但是， `boost::numeric_cast` 验证了在不改变数值的情况下转换是否能够发生。 前面给的应用例子，转换不能发生，因而由于 `0x10000` 太大而不能存储在 `short` 类型的变量上，而抛出 `boost::numeric::bad_numeric_cast` 异常。

严格来讲，抛出的是 `boost::numeric::positive_overflow` 类型的异常，这个类型特指所谓的溢出(overflow) - 在此例中是正数。 相应地，还存在着 `boost::numeric::negative_overflow` 类型的异常，它特指负数的溢出。

```
#include <boost/numeric/conversion/cast.hpp>
#include <iostream>

int main()
{
  try
  {
    int i = -0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << std::endl;
  }
  catch (boost::numeric::negative_overflow &e)
  {
    std::cerr << e.what() << std::endl;
  }
}
```

- 下载源代码

Boost.NumericConversion 还定义了其他的异常类型，都继承自 `boost::numeric::bad_numeric_cast` 。因为 `boost::numeric::bad_numeric_cast` 继承自 `std::bad_cast`，所以 `catch` 处理也可以捕获这个类型的异常。